YIIHAW
An aspect weaver for .NET

IT University
of Copenhagen

Rasmus Johansen **Stephan Spangenberg**

Supervisor:
Peter Sestoft

IT University of Copenhagen
28/02/2007

Abstract

This thesis has examined various aspect weavers that exist for the .NET platform with the purpose of investigating their applicability for generating specialized programs and weaving performance-critical applications. Neither of these aspect weavers turned out to be usable for these purposes, as they added too much runtime overhead or lacked the basic features needed for implementing the proper aspects. Aspect.NET performed better than the other weavers in terms of efficiency of the generated code, but with its lack of support for introductions and interceptions of instance methods it turned out to be too limited to be of any use.

We have proposed a different implementation for an aspect weaver that uses inlining of the IL-instructions when applying advice methods to a target assembly. Using this approach, many of the unnecessary constructs that other aspect weavers add to the generated assemblies can be avoided, such as assembly references and copies of the advice methods. This means that the program structure defined by the user in the target assemblies are completely maintained once the weaving has been performed.

The implemented prototype supports various features required for implementing advanced AOP-constructs, such as introductions and typestructure modifications. The aspect language provides a simple and intuitive programming model for implementing aspects. The weaver performs typechecking on all constructs, guaranteeing that only valid assemblies are generated, i.e. assemblies that are verifiable by the Common Language Runtime.

We consider the prototype to be highly usable for constructing such program-generators that cannot be implemented using any of the aspect weavers examined during this thesis. Empirical tests show that the weaver prototype does not introduce any runtime overhead in the generated assemblies, making it suitable for applying aspects to performance-critical applications.

Contents

1 Introduction **6**

2 Problem definition **7**
2.1 AOP in modern programming . . . 7
2.1.1 Pointcuts and join points . . . 8
2.1.2 Interceptions . . . 9
2.1.3 Introductions . . . 10
2.1.4 Typestructure modifications . . . 10
2.2 The need for a high-performing aspect weaver . . . 11
2.2.1 Case study: C5 collection library . . . 11
2.3 Insufficiency of existing aspect weavers . . . 12
2.3.1 Existing aspect weavers . . . 12
2.3.2 Defining the tests . . . 13
2.3.3 Test setup . . . 17
2.3.4 Test results . . . 17
2.4 Preliminary goals for a high-performing aspect weaver . . . 19

3 Problem analysis **21**
3.1 Inside Aspect.NET . . . 21
3.1.1 Intercepting a method call . . . 21
3.1.2 Advice return type . . . 22
3.1.3 Using arguments . . . 22
3.1.4 The lessons learned . . . 23
3.2 Binding mode . . . 25
3.3 Applying aspects . . . 26
3.3.1 Source code weaving . . . 26
3.3.2 Direct advice invocation . . . 28
3.3.3 Inlining advice . . . 29
3.4 Implementing aspects . . . 31
3.4.1 Introductions . . . 31
3.4.2 Interceptions . . . 32
3.4.3 Typestructure modifications . . . 37
3.5 Pointcut specification . . . 38
3.5.1 Writing the pointcuts . . . 38
3.5.2 The pointcut language . . . 39
3.5.3 Wildcards . . . 39
3.5.4 Complex expressions . . . 39
3.6 Summary . . . 40

4 Common Intermediate Language **41**
4.1 What is CIL? . . . 41
4.1.1 Assembly, modules and metadata . . . 41
4.1.2 Instructions and operands . . . 42
4.1.3 Flow control . . . 42
4.1.4 Datatypes . . . 43
4.1.5 The stack . . . 43
4.1.6 Exception handling . . . 43

5 Working with CIL **46**
5.1 Microsoft Phoenix . . . 46
5.1.1 Is Phoenix the right choice? . . . 47
5.2 Cecil . . . 47
5.2.1 Is Cecil the right choice? . . . 47

6 The complexity of advice inlining **48**
6.1 Advice syntax . . . 48
6.2 Invoking Proceed . . . 50
6.2.1 Handling void . . . 50
6.2.2 Injecting the target method's body . . . 51
6.3 Merging local variables . . . 53
6.4 Mapping IL-instructions . . . 53
6.5 Checking references . . . 53
6.5.1 Handling references to constructs outside the aspect assembly . . . 54
6.5.2 Handling references to constructs in the aspect assembly . . . 55
6.5.3 Handling references to local constructs . . . 55
6.6 Referring to the declaring type of the target method . . . 56
6.6.1 Handling calls to *GetTarget()* . . . 58
6.6.2 Accessing constructs of the declaring type . . . 58

7 Pointcut specification **59**
7.1 Defining the pointcut language . . . 59
7.1.1 The pointcut file . . . 59
7.1.2 Targeting constructs . . . 59
7.1.3 The pointcut language . . . 61
7.2 Scanning the input file . . . 63
7.2.1 Identifying keywords and special tokens . . . 63
7.3 Parsing the pointcuts . . . 64
7.3.1 Storing the pointcut statements . . . 65
7.3.2 Handling errors . . . 65

8 Identifying targets and aspects **66**
8.1 Locating constructs in the target and aspect assemblies . . . 66
8.2 Locating the proper advice method . . . 66
8.2.1 Invocation kind of the advice methods . . . 67

9 The weaving process **68**
9.1 Introducing constructs . . . 68
9.1.1 The need for a two-pass approach . . . 69
9.1.2 Storing constructs in a mapping-table . . . 69
9.2 Modifying the typestructure . . . 71

9.2.1 Verifying the availability of the basetypes and interfaces . . . 71
9.2.2 Checking the definition of methods and properties . . . 71
9.2.3 Updating references . . . 71
9.3 Handling interceptions . . . 72
9.3.1 The join point API . . . 72
9.3.2 Merging the targets and advice . . . 73
9.4 Using YIIHAW . . . 75

10 Partial functional testing **76**
10.1 The test framework . . . 76
10.2 The testcases . . . 77
10.2.1 A test sample . . . 77

11 Measuring the runtime performance **78**
11.1 Test setup . . . 78
11.2 Comparing YIIHAW to code written by hand . . . 78
11.3 Implementing a generator for a collection library . . . 78
11.3.1 Test scenario . . . 79
11.3.2 Test results . . . 80

12 Evaluation **81**
12.1 Runtime performance of the generated assemblies . . . 81
12.2 The aspect language . . . 83

13 Future work **85**
13.1 Support for further introductions . . . 85
13.2 Handling typestructure modifications . . . 85
13.3 Further possibilities in the join point context API . . . 85
13.4 Compatibility with older releases of .NET . . . 85
13.5 Accessing generic parameters . . . 86
13.6 Support for generics in the pointcut language . . . 86
13.7 YIIHAW on the web . . . 86

14 Conclusion **87**

A Usage guide for YIIHAW **92**

B Pointcut grammar **101**

C Short form notation for the pointcut language **103**

D Class diagram **104**

E Source code for tests - Basecode **105**

F Source code for tests - Coded by hand **109**

G Source code for tests - AspectDNG **113**

H Source code for tests - Aspect.NET **116**

I Source code for tests - NKalore **118**

J Source code for tests - Rapier LOOM **122**

K Source code for tests - YIIHAW **129**

L Source code for collection tests - AspectDNG **131**

M Source code for collection tests - YIIHAW **141**

N Source code for collection tests - Coded by hand **149**

O Source code for collection tests - Basecode **160**

P Source code for collection tests - Test program **168**

Q Partial functional testing overview **170**

R Source code for partial functional testing **173**

S Source code for YIIHAW - API **216**

T Source code for YIIHAW - Exceptions **219**

U Source code for YIIHAW - Output **221**

V Source code for YIIHAW - Pointcut **229**

W Source code for YIIHAW - Controller **258**

X Source code for YIIHAW - Weaver **297**

Chapter 1

Introduction

Aspect Oriented Programming (AOP) is a paradigm that has received a lot of attention during the last couple of years. This is due to the nature of AOP that directly addresses some of the shortcomings in traditional programming paradigms, such as object oriented programming (OOP) and procedural programming. These programming paradigms do not fully support the handling of so-called cross-cutting concerns and thus often fall short when expressing these types of interests.

AOP offers features for implementing and handling cross-cutting concerns (named *aspects*) via a program called an aspect weaver. The Java community has been leading the field of AOP for some time, primarily due to AspectJ [6] which was one of the first aspect weavers available and is probably the most well-known weaver today. So far, AOP has not received the same amount of focus in the .NET world. Naturally this affects both the quantity and quality of available weavers for the .NET platform. Most of these weavers are created as small "hobby-projects" that do not receive any commercial support, unlike AspectJ which is developed and distributed as part of the Eclipse project. This lack of commercial support means that these products do not achieve a wide usage in the .NET community and are thus quickly forgotten and abandoned. Obviously, this also affects the maturity of these products, which results in limited support for certain features, like generics, low-level pointcut specification (see section 2.1), typestructure modifications, etc. Another problem, one that is important for this thesis, is the performance penalty that is often incurred when using existing weavers. High performance penalties when intercepting methods are not unusual. Obviously, this does not make the use of AOP viable in performance-critical applications. This thesis will investigate the causes for these performance penalties and discuss how they can be dealt with. To support these discussions a working prototype of an aspect weaver will be developed that tries to resolve the performance issues of existing weavers.

This thesis is highly motivated by our previous project: "Generation of specialized collection libraries" [3]. That project examined various techniques that could be used for implementing a generator for creating customized versions of the C5 collection library [4]. The overall conclusion of that project was that AOP offered a high degree of code reusablity and provided a great reduction of the complexity of the generator. Unfortunately the generated code performed very poorly (in regard to CPU time) due to an inappropiate implementation of the aspect weaver (AspectDNG [5]). Thus, optimal performance was presumably not considered a high priority when that aspect weaver was designed. An aspect weaver without (or with very little) performance overhead would be very useful for implementing such a generator [3].

Chapter 2

Problem definition

This chapter presents the problem at hand in greater detail. The following section briefly describes what AOP is and how it can be used. This description presents the generel terms and usage of AOP - it is not targeted specifically at the problem domain of this thesis. Having presented the concept of AOP, some of the existing aspect weavers for .NET will be presented with focus on their runtime performance.

2.1 AOP in modern programming

A key issue in modern programming is the design of a maintainable and extensible program structure that allows replacement and modification of parts of the program, so that program behaviour can be easily expanded or altered as needed. Obviously this requires the program structure to be designed for handling such changes. *Separation of concerns* [13] is a classical programming principle that suggest encapsulation of concerns (points of interest) into self-describing and more or less autonomous entities, such as a classes, modules, components, etc. The motivation for this principle is that such encapsulation result in a weaker coupling between different concerns in the program, ensuring that the dependencies between different parts of the program are reduced. Creating a program structure with weak coupling between different concerns is a fundamental goal as it allows great flexibility when it comes to program modification. Concerns that are tangled within each other and scattered over the source code are cumbersome to handle when it comes to refactoring the code. A well-designed program structure ensures that modifications in program behaviour can easily be adapted to fit within the existing program.

Some concerns cannot easily be encapsulated or localized into separate entities or program constructs. Such concerns are said to be *cross-cutting concerns*. Logging is the classical example of a cross-cutting concern: Often statements such as *Logger.Write("entering method XXX")* are inserted at various locations in the source code to trace program execution through the log. Even simple applications might contain hundreds of such statements that are simply repeated over and over. Obviously such repetition of code is tedious, not to say error-prone. AOP directly targets such problems by allowing you to write the concern once and then apply it multiple times - either to the compiled binary or to the source code. In the logging example described above a single implementation of the *Logger.Write(...)* statement would do - this implementation can then be inserted at various locations in the program through the use of an *aspect weaver*. An aspect weaver is a program that allows merging multiple source code files or compiled binaries into one entity. This combined entity contains the original implementation along with the implemented concerns inserted at specified locations.

The main benefits that AOP offers is code reduction and code overview: A concern need to

be implemented only once in order to be applied at many locations. Thus, one does not need to write the same lines of code over and over at different locations in the source code. This issue is refered to as *quantification* [8]. Another central issue in AOP is that of *obliviousness* [8]: The use of AOP should be transparent in the programming of the *target code* (the part of the implementation where concerns should be applied), i.e. no special preparations should be assumed. Thus, no special constructs should be created in the target code prior to applying the implemented concerns.

2.1.1 Pointcuts and join points

Being able to implement concerns once and apply them multiple times is a key objective in AOP. To support this feature obviously requires some means of specifying where to apply the concerns. This is done through the use of *join points* and *pointcuts*. A join point describes a particular point in execution of the code. Most often, a join point directly matches a specific location in the source code, e.g. a specific method, but join points can also describe a particular control flow event that does not directly correspond to a particular line in the source code. For instance, one might define a join point that matches all calls to method *Foo()* from any instance of class *Bar*. This join point cannot be described by a particular line in the source code as the join point should only apply to method calls to *Foo()* that comes from *Bar* - calls to *Foo()* from any other type of object should not be matched by this join point. This join point thus matches a specific control flow in the execution of the program.

Pointcuts have a close relationship with join points: A pointcut is a pattern used to define one or more join points. An example of the use of pointcuts is shown below.

```
pointcut loggingMethods(): call(* *.set*(..));

before(): loggingMethods()
{
    Logger.write("entering " + thisJoinPoint);
}
```

Figure 2.1: A pointcut in AspectJ.

Figure 2.1 shows how to apply a logging concern at multiple locations. First, a pointcut named *loggingMethods* is defined. This pointcut matches all calls to methods that return any type (the first '*'), is located in any class (the second '*'), whose name starts with 'set' and takes any number of arguments (the '..' part). Once the pointcut is specified it can be used when implementing the concern. In figure 2.1 the keyword *before* is used, which states that the implementation should be applied immediately before the call to any method that matches *loggingMethods*. Thus, prior to executing such a method the logging concern is executed. This particular concern uses what is called *join point context*: The variable *thisJoinPoint* is a special variable introduced by AspectJ, that gives access to context information about the current join point, e.g. the signature of the original method (the method being intercepted) as it is used in the example above. The *thisJoinPoint* variable can also be used to retrieve information about the return value and arguments of the original method.

Dynamic pointcuts

The pointcut shown above can be determined statically, which means that the all join points that match the pointcut can be identified at weave-time. Some aspect weavers also support so-called *dynamic pointcuts*, i.e. pointcuts that describe events not known until runtime. For instance, AspectJ [6] defines a dynamic pointcut event called *cflow*. This allows the user to specify the identity of the caller when defining an interception. This is shown in the small sample below.

```
call(void ClassA.bar()) && cflow(call(void ClassB.foo()))
```

This pointcut matches all calls to *ClassA.bar* that occurs from *ClassB.foo*. This includes both direct and indirect calls - if *ClassB.foo* calls another method which in turns calls *ClassA.bar* this matches the pointcut as well. Calls to *ClassA.bar* from all other methods are not matched and are thus not intercepted. Dynamic pointcuts, such as this one are not always determinable at weave-time, as the cflow might match a method in another assembly that is not available at weave-time. AspectJ thus places so-called *dynamic residue* in the generated code. Dynamic residue is simply some code that performs a runtime check to see if a match is found and then applies some code specified by the user. This approach thus defers some parts of the weaving until runtime.

2.1.2 Interceptions

Interception is the process of adding and modifing parts of the target code by changing the control flow. The code presented in figure 2.1 is an example of an interception: The logging concern intercepts the methods matching the pointcut and thus alters the control flow of the original program. The specific code that constitutes the concern of interest is refered to as *advice* and is implemented in a separate method (referred to as an *advice method*).

Interceptions are not restricted to methods only. Many aspect weavers allow you to intercept properties, constructors, field access (read/write), etc., thus allowing one to alter the control flow as needed. When talking about interceptions one usually distinquish between three kinds: *Around*, *before* and *after*. These can be further divided into two types: *call* and *body*.

Around

Around interception is one of the most fundamental AOP constructs and is supported by most implementations. It basically allows modifying the control flow of an application by replacing one or more target operations. The target operation will most often be a method in the target code, but it could just as well be a property, a constructor or a field access. The operation to target is determined via a pointcut.

Basically, two types of *around interception* exist: *around body* and *around call*. An *around body interception* allows replacing the body of an existing target (which means that you can actually discard all of the original body). An *around call interception* does not modify the body of the method being intercepted, but instead modifies all calls to this method so that the advice method is invoked instead. For both cases the original method can be invoked at some point in the advice. Throughout the rest of this report, when referring to *around interception* we mean *around body interception* unless explictly stated otherwise.

Even though *around interception* can be used to remove code from a method, most oftenly an around interception is used to add additional code to a target. Most implementations support

invoking the original target from within the advice. This is usually done through a method named *Proceed()* or similar. This allows you to write the logic of the advice (the additional code) and then invoke the original target at a suitable time, e.g. as the last instruction in the advice.

Before

Before is another type of interception. It basically allows intercepting a method by executing the advice immediatly *before* returning control to the original method. As with around interception, two types of *before interception* exist: *before call* and *before body*. Using the former, all invocations of the target method (the method that is being intercepted) are replaced with a call to the advice. Using the latter, the advice is inserted into the target body. Unlike *around interception* the programmer does not need to explicitly invoke a *Proceed()* method - the original method is always executed.

A *before interception* can be implemented using an *around interception* instead: An *around interception* that ends with the invocation of the *Proceed()* method simulates the effect of a *before interception*. From a logical perspective they do the same (that is, they execute the same code). However, as *around interception* usually suffer from a greater runtime overhead than *before interception* [9], many implementations directly supports *before interception*, which can often be implemented more efficiently.

After

After interception is similar to *before interception*, except that the advice is executed *after* the original method has completed. Again, *after interception* can be simulated using *around interception*, but this usually introduces a greater runtime overhead [9]. As *before interception*, *after interception* can be specified to be either *after call* or *after body*.

2.1.3 Introductions

Being able to intercept control flow in an existing program is a major benefit when dealing with cross-cutting concerns. However, sometimes you also need some way of inserting new constructs (methods, fields, properties, classes, etc.) into an existing program. This process is referred to as *introduction*. By introducing new constructs you are modifying the program structure of the existing program so that it reflects the concerns of interest. This is useful when you need to expand or modify the behaviour of an existing program.

2.1.4 Typestructure modifications

Some aspect weavers support the use of *typestructure modifications*, which means that the weaver can alter the typestructure of an existing program. This can usually be done in two ways: Change the superclass of a class in the target assembly or make a class implement one or more interfaces. This process is not as widely used as *introductions*, but it can still be very useful in some cases. For instance, when using AOP for generating programs one often need to add behaviour to existing classes. It often makes sense to provide access to this behaviour through one or more interfaces in order to achieve a low coupling between different entities in the target assembly.

2.2 The need for a high-performing aspect weaver

Object-oriented programming is a very expressive programming paradigm that allows easy encapsulation, modularization and code reuse. However, as mentioned in section 2.1, OOP often falls short when it comes to handling cross-cutting concerns. Expressing such concerns often requires a great deal of code scattering and code tangling, resulting in fragmented and repetitive source code. This introduces the need for an aspect weaver for the .NET framework. Such an aspect weaver should support the features required by the programmer for handling cross-cutting concerns without introducing unnecessary runtime overhead. In many programs AOP is useless if the concerns cannot be implemented in an efficient manner, as this would simply outweigh the advantage of using AOP in the first place. An aspect weaver that can be used for these kind of programs is the primary focus for this thesis. The following case study depicts a scenario where the use of an inefficient aspect weaver would simply not be suitable.

2.2.1 Case study: C5 collection library

The C5 collection library [4] is a general-purpose collection library for .NET that provides support for datastructures not found in the standard collection library distributed as part of the .NET framework. In an earlier project [3], we examined how to make a generator for a small sample of this collection library that can be used for customizing it for specific purposes. This generator allows configuring which datastructures should be included in the library (linked lists, array lists, etc.) and what features these datastructures should support (enumeration, event-handling, etc.). A key concern is that the use of a generator should not introduce any overhead in the library, as that would neglect the use of a generator.

As the implementation of C5 consists of roughly 27,000 lines of code [4], a well-structured approach was needed. Simply using preprocessing directives (`#if <flag> ... #endif`) on the source code was not a suitable solution, as it would be too comprehensive to generate and maintain preprocessor statements for all optional features in the code. Instead a different approach was suggested: The optional features were implemented as advice and were separated from the rest of the code. This resulted in a huge reduction in the complexity of the generator, due to the following:

- It allowed reusing the features at many locations (through the use of pointcut specifications).
- It made precompiling the optional features possible, which ensured that an early type-checking was achieved.
- It allowed separating the optional features from the configuration of where these features should be applied. The only thing that the generator would need to perform at runtime was the generation of the configuration file and invoking the aspect weaver.

Thus, from a structural and conceptual point of view the use of AOP seems feasible for implementing such a generator. However, a main conclusion of the project was that the choice of which aspect weaver to use plays a significant role in regards to the amount of overhead that is introduced. Using AspectDNG [5] the overhead incurred was as much as 5,000%! This obviously calls for a solution that offers the basic AOP features needed, but without introducing any overhead.

2.3 Insufficiency of existing aspect weavers

This section will present some of the aspect weavers for .NET and discuss why we believe that they are not yet suited for handling concerns in a resource efficient manner. This discussion will primarily be based on a number of tests that will be performed on the aspect weavers in question. The motivation for these tests will be described as well.

2.3.1 Existing aspect weavers

Various aspect weavers exist for the .NET framwork. As it is impossible to test all of them, we have decided to pick the ones that we consider to be the most popular. These weavers will be used as a foundation for our tests.

AspectDNG

AspectDNG [5] is an open source aspect weaver that uses static evaluation[1] of all aspects. When intercepting a method, advice is inserted into the target assembly as a method and the target method is modified so that it invokes the advice method. After the weaving has taken place the target assembly is completely self-contained, i.e. the target assembly is not dependent on the advice assembly or any other assemblies.

AspectDNG supports *around interceptions* as well as typestructure modifications and introductions. Two choices exist when it comes to defining the pointcuts: One can either annotate the advice methods and classes using .NET attributes or one can use an external XML configuration file. Using the latter means that access to the source code is not required.

Aspect.NET

The Aspect.NET [15] weaver is developed at St. Petersburg University in Russia and is supported by Microsoft Research. Like AspectDNG, it is a static weaver. It uses Microsoft Phoenix [18] for manipulating the target assembly. In the current version (2.0 at the time of writing) the weaving is done by inserting new method calls to the advice methods in the aspect assembly at the join points specified by the pointcuts. When using *around interceptions* the original method call is removed (effectively resulting in an *around call interception*).

As can be seen in figure 2.2 and 2.3 the functionality in the current version is limited to only using static methods as advice and only targeting static methods when using *around interceptions*. There is no support for introductions and only methods can be the target of pointcuts.

Pointcuts can be specified either through the use of a specially developed Aspect.NET.ML metalanguage or by creating attribute classes. In the former case the metalanguage file will be automatically translated to an attribute class.

To use Aspect.NET the user needs to have Visual Studio.NET installed as the weaving is done through a plugin inside this IDE. When using this tool it is possible to see all the join points in the source code and to choose whether or not to include specific join points in the weaving.

[1] Static evaluation means that all targets are identified only by looking at the compiled assemblies. No runtime checks are performed. This subject will be elaborated upon in section 3.2.

Rapier LOOM

Rapier LOOM[10] is a dynamic weaver[2] developed at the Hasso Platner Institut at the Postdam University in Germany. There is no public documentation available about the inner workings of the weaver, however we know that an earlier version of the weaver used reflection to create proxies for the target classes [11]. After using a debugger tool on a weaved assembly, we believe that the weaver still works by using reflection and proxies. The pointcut language in the weaver is based upon custom made attributes, which are set on the advice methods and the aspect classes.

As shown in figure 2.3 there are a lot of demands on the target, when using Rapier LOOM. As the weaving is done through a factory method, it is only possible to do weaving on objects and not on classes (i.e. static methods are not supported). Using factory methods also means that the creation of objects should no longer be done by using *new*, but instead by using a special factory method. This means that one can use both weaved and unweaved versions of the same objects in the program if that is needed. Rapier LOOM requires that the target methods has to be virtual or defined in an interface.

Some of the more advanced features in Rapier LOOM include the possibility of intercepting exception throwing and introducing interfaces together with matching methods upon a target object.

NKalore

NKalore [17] is a static weaver based on the Mono C# compiler [16]. Advice and pointcuts are written directly inside the source files using a special metalanguage that extends C#, which means that access to the source code is a requirement for NKalore to work. To compile the source files, the included compiler must be used. This compiler takes care of the weaving as well. The aspects are thus compiled directly into the assembly. There is no support for modifying an existing assembly.

NKalore only supports interceptions (*before*, *after*, *around* and *exceptions throwing*). Introductions and typestructure modifications are not supported.

A summary table comparing the features of the aspect weavers presented during this section can be seen in figure 2.2 and 2.3.

2.3.2 Defining the tests

In order to identify some of the shortcomings that exist when using typical AOP constructs in existing weavers, a number of tests will be performed. These tests are not meant to thoroughly exercise every possible use of aspects as this would require a project on its own. The purpose is to determine the net effect of applying some of the most frequently used constructs, such as interceptions, method introductions and inheritance modifications. The tests that will be used are somewhat ad hoc; the key intention is to identify the direct consequences of using an aspect weaver. The tests do not directly reflect real-life usage of aspects. The reason for this is simple: Real-life applications do not give a clear indication of the overhead of using an aspect weaver, because there is simply too much "noise". A typical application might implement 5% of its concerns via aspects, meaning that the overhead would only constitute a very small percentage of the total time and memory consumption. This obviously makes it more difficult to directly

[2] A dynamic weaver applies the advice at runtime, just before the classes and methods in question are loaded into memory. This subject will be elaborated upon in section 3.2.

Aspect weaver	.NET version	Introductions	Interceptions	Basetype modif.	Interface impl.
AspectDNG	1.1 2.0	methods fields classes interfaces	around body/call field read/write	+	+
Aspect.NET	1.1 2.0	–	before call after call around call	–	–
Rapier LOOM	1.1 2.0	methods interfaces	before body after body around body after returning after throwing	–	+
NKalore	1.1 2.0	–	before body after body around body	–	–

Figure 2.2: Aspect weaver comparison chart.

Aspect weaver	Pointcut specification	Demands on target	Binding mode	Source code required
AspectDNG	attributes config file	–	static	–
Aspect.NET	attributes metalanguage	If using instead, target method must be static	static	+
Rapier LOOM	attributes	Methods must be virtual or part of an interface. Objects need to be created by LOOM factory and aspects must be instantiated by target.	dynamic	+
NKalore	metalanguage	Must use included compiler (Mono)	static	+

Figure 2.3: Aspect weaver comparison chart (continued).

measure the consequences of using an aspect weaver. The tests that we propose are somewhat more synthetic; they consist of one or more simple AOP constructs that are simply repeated a fixed number of times. These tests can be considered as a "worst-case" scenario for the weavers, as they are designed to make as much use of the aspects as possible. This means that any overhead added by the weavers will be very significant in these tests, as nearly all CPU cycles are used on executing the aspects.

We compare the results of these tests to an implementation coded by hand. This implementation is used as a reference when measuring the overhead of the weavers and can be considered as the optimal implementation, as it is naturally not dependent on any aspect weaver, which

means that no overhead exist in this implementation. All aspect weavers use the same target assembly for applying the aspects. The tests are presented below. The implementation of the tests can be seen in Appendix E - J. The results are presented in section 2.3.4.

Test 1: Around interception

The purpose of this test is to determine the cost of applying *around interceptions*. Consider a class, *Tester*, that contains an instance method, *ToBeIntercepted()*, and a static method, *Main()*, as shown in figure 2.4.

```
class Tester
{
  public static Random r = new Random();

  public void ToBeIntercepted()
  {
    r.Next(); // get a random integer
  }

  public static void Main(string[] args)
  {
    Tester t = new Tester();

    for (int i = 0; i < 10000000; i++)
      t.ToBeIntercepted();
  }
}
```

Figure 2.4: The class *Tester*.

```
public void Advice()
{
  r.NextDouble(); // get a new random number using the static Random object
  Proceed(); // invoke the original method
}
```

Figure 2.5: Advice for the around interception (pseudo-code).

The *Main()* method simply invokes the *ToBeIntercepted()* method 10,000,000 times. These two methods constitute the target code of the test. Using an around interception, the advice code shown in figure 2.5 should be added to the *ToBeIntercepted()* method. This advice simply generates a random floating point number and invokes the original method (written as "Proceed()" above). The concrete syntax for implementing this kind of advice depends on the aspect weaver used.

One might argue that implementing the *ToBeIntercepted()* as an empty method would simplify the tests even more. However, this might also trigger various code optimzations in either the compiler or in some of the aspect weavers. The generation of random numbers were chosen as they are not subject to any kind of optimizations, as there is no way to pre-determine a random number or determine the effect of invoking the *NextDouble()* method.

Test 2: Before interception

Applying a *before interception* might yield better results than using an *around interception* (refer to section 2.1.1). In order to determine the effect of applying a *before interception*, a test similar to the one described above is performed using *before interception*, but without invoking *Proceed()*. Apart from that the tests are similar.

Test 3: After interception

A test similar to the one described above is also performed for *after interceptions*.

Test 4: Around interception with method argument

Most aspect weavers support fetching method arguments during interception. This allows you to use arguments passed to the original method within your advice. In order to test the effect of fetching method arguments, the test presented in Test 1 is slightly modified for this test, as can be seen in figure 2.6.

```
public void ToBeIntercepted(Random r)
{
  r.Next(); // get a random integer
}

public void Advice(Random r)
{
  r.NextDouble(); // get a new random number using the Random argument
  Proceed(); // invoke the original method
}
```

Figure 2.6: The *ToBeIntercepted()* method and the advice (pseudo-code).

The *ToBeIntercepted()* method now takes an instance of *Random* as argument. This instance is used within the *ToBeIntercepted()* method and the advice.

Test 5: Around interception on a static method

The previous tests have all been based on intercepting instance methods. To determine whether intercepting static methods differs from interceptng instance methods, a test will be performed similar to that of Test 1, with the exception that *ToBeIntercepted()* and the advice are now declared as being static.

Test 6: Introducing a new method

Some concerns require introducing new constructs (methods, fields, classes, etc.) into the target assembly. Naturally, these introductions should not cause severe performance drops. To test the effect of introducing new constructs into the target assembly, the following test is performed:

An empty class, *Tester*, is defined. Through the use of the aspect weavers, a method, *GetNextInt()* should be inserted into this class. This method is shown in figure 2.7.

In a separate assembly, another class is defined that invokes the *GetNextInt()* method 10,000,000 times and measures the elapsed time. This requires that the second assembly refers to the assembly containing the *GetNextInt()* method. Using a separate assembly for invoking the

```
public void GetNextInt()
{
    r.Next();
}
```

Figure 2.7: The *GetNextInt()* method.

GetNextInt() might seem cumbersome at first. Another approach would be to simply introduce the new method and invoke this method by intercepting an existing method in the target code. However, this makes it difficult to measure the mean effect of introducing a new method, as the measurement would also include the time taken to execute the advice. To avoid this, the invocation of *GetNextInt()* is implemented in a separate assembly, where the invocation can be coded by hand.

Test 7: Typestructure modification

The last test measures the effect of changing the inheritance structure. An abstract class, *SuperClass*, is defined, containing a single method, *GetNextInt()*. Two subclasses are defined, *SubA* and *SubB*, that both implement the *GetNextInt()* method. The implementation for both classes are similar to the implementation of *GetNextInt()* in Test 6. Having defined a superclass and two subclasses, a class, *Tester*, is defined that inherits *SubA*. The *Main()* method of this class simply invokes the *GetNextInt()* method (which is implemented in *SubA*) 10,000,000 times and measures the elapsed time. By the use of the aspect weavers the inheritance structure should now be modified, so that class *Tester* extends *SubB* instead of *SubA*. This modification means that the invocation of *GetNextInt()* now refers to the implementation defined in *SubB*.

2.3.3 Test setup

For all tests, execution time is measured (in milliseconds) as an average of 25 testruns. For each test the average deviation in execution time is calculated as well. The average deviation can be used to determine the degree of uncertainty in the measurements.

All tests were performed on a machine with the following specifications:

- Pentium 4 Mobile 1.2 GHz
- 512 MB RAM
- Windows XP SP 2
- .NET Framework 2.0.50727

All tests are compiled as *release builds* with *code optimization* turned on.

2.3.4 Test results

The results of the tests are shown in figure 2.8. The first thing to notice when looking at the results is the tremendous performance drop in Test 1 when using AspectDNG, NKalore and Rapier LOOM. In all cases, a huge increase in the execution time can be seen: Rapier LOOM is around 65 times slower than the reference implementation and NKalore is more than 460 times slower!. Recall that Test 1 used *around interception* on an instance method. In order to identify the causes for these performance drops, the weaved assemblies of AspectDNG and

Aspect weaver	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7
Coded by hand	716 (1.00) 0.25%	716* (1.00) 0.25%	716* (1.00) 0.25%	716 (1.00) 0.17%	719 (1.00) 0.24%	604 (1.00) 0.37%	747 (1.00) 0.53%
AspectDNG	260119 (363) 0.31%	–	–	–	258414 (359) 0.18%	605 (1.00) 0.30%	744 (1.00) 0.16%
Aspect.NET	–	743 (1.04) 0.43%	765 (1.07) 0.31%	815** (1.14) 0.10%	791 (1.10) 0.19%	–	–
NKalore	334464 (467) 0.42%	83323 (116) 0.56%	83250 (116) 0.63%	364443 (509) 0.50%	322408 (449) 0.42%	–	–
Rapier LOOM	46969 (65.6) 0.32%	2698 (3.77) 0.23%	2904 (4.06) 0.23%	48438 (67.6) 0.11%	–	39150 (54.5) 0.14%	–

Figure 2.8: Test results. The first number in each cell is the average execution time in milliseconds. The second number (in parentheses) specifies the execution time as a factor of the reference implementation. The third number is the average deviation (in percent) from the average execution time for each testrun.

– = The aspect weaver do not support the features required for implementing this test.
* = The results of Test 1 are simply repeated in Test 2 and Test 3, as they are similar to Test 1 when coded by hand.
** = This test is slightly modified from the original test decription, as the *ToBeIntercepted()* method is made static as Aspect.NET cannot handle instance methods when using *around interception*.

NKalore were examined[3] using the Intermediate Language Disassembler (*ildasm*). It turned out that both weavers made heavy use of reflection for making join point context available and for invoking the original method. Furthermore, compared to the reference implementation, a huge increase in the number of opcodes[4] were identified: For AspectDNG, the weaved version of the *ToBeIntercepted()* method now consisted of a total of 19 opcodes, including 5 method calls and 1 object instantiation. NKalore used a total of 27 opcodes (7 method calls and 1 object instantiation). These figures are to be compared to the reference implementation, which implements the *ToBeIntercepted()* method using only 7 opcodes, including 2 method calls and 0 object instantiations. This huge increase in the number of opcodes and method calls obviously has a great impact on the execution time.

Implementing the *ToBeIntercepted()* method using *before* and *after interceptions* yields much better results for NKalore and Rapier LOOM as shown in figure 2.8 (Test 2 and Test 3). For NKalore the runtime overhead is reduced by a magnitude of 4 compared to Test1. However, the generated code is still more than 100 times slower than the reference implementation. Rapier LOOM actually achieves fairly good results in test 2 and test 3: Compared to test 1 the runtime

[3]Unfortunately, it is impossible to examine the generated code of Rapier LOOM, as it applies all aspects at runtime.

[4]An *opcode* is an Intermediate Language instruction, such as *add* or *call*, which gets JIT-compiled and executed by the Common Language Runtime (CLR). Chapter 4 will give greater insights into this concept.

overhead is improved by a factor 17, although the generated assemblies are still around four times slower than the reference implementation.

Looking at the results in Test 2 and Test 3, the smallest overhead is achieved using Aspect.NET. Aspect.NET is only around 30 and 50 milliseconds slower than the reference implementation in these tests, which roughly corresponds to 4% and 7%. Using *ildasm* on the Aspect.NET assemblies, it can be seen that the advice is simply implemented as a method (called "test2Aspect") in a separate assembly. Aspect.NET only makes two modifications to the original assembly: The advice assembly is added as a reference and a method call to the advice is inserted either before (Test 2) or after (Test 3) the call to *ToBeIntercepted()*. This approach seems to perform a lot better than the techniques used in any of the other weavers.

All aspect weavers seem to experience a small performance drop when using method arguments (compare to the results of Test 1). For NKalore the result of this test is about 9% slower than test 1, which means that the generated assembly is now more than 500 times slower than the reference implementation! Aspect.NET is around 7% slower in this test compared to Test 2 and 3. Keep in mind that this test is implemented using static methods in Aspect.NET, as this weaver does not support intercepting instance methods. This might have an effect on the results. Rapier LOOM has the lowest performance drop with only 3% compared to Test 1. AspectDNG does not support using method arguments when intercepting methods and could thus not complete test 4.

The use of static methods instead of instance methods do not seem to make any difference for any of the weavers. There is practically no difference between Test 5 and Test 1 in this regard[5]. NKalore is actually a little bit faster than test 1, but the difference is too small to make any general assumption based on these results.

Introducing methods (Test 6) do not seem to cause any overhead for AspectDNG: The execution time of the generated assembly are similar to the reference implementation. Only Rapier LOOM seem to add some runtime overhead (Rapier LOOM is more than 50 times slower than the reference implementation). However, the dynamic nature of Rapier LOOM makes it somewhat difficult to test the execution time of an introduced method in a fair manner: The static weavers introduce the new method into an existing assembly. This method can then be invoked from another assembly that is coded by hand (as we have done in this test). This is not possible with Rapier LOOM as it introduces the method runtime, i.e. the method cannot be invoked statically from another assembly. Thus, invocation of the newly added method is performed runtime (just after the introduction) via a proxy. This might have an impact on the result.

Only AspectDNG was able to implement Test 7. As expected, the inheritance modification did not cause any measurable overhead.

2.4 Preliminary goals for a high-performing aspect weaver

Looking at the test results the overall conclusion is clear: Aspect.NET achieves by far the best results in terms of runtime overhead: For all tests, the assemblies generated by Aspect.NET lies within 14% of the reference implementations. Although these results are good, we do not consider Aspect.NET to be viable for efficiently handling cross-cutting concerns, as we believe

[5]These tests are comparable as they only differ in the use of instance methods (Test 1) or static methods (Test 5).

that the use of AOP should not add any runtime overhead to the assemblies. Looking at the assemblies generated by Aspect.NET, it can be seen that a lot of extra instructions are added that we believe can be avoided. We do thus not consider the output of Aspect.NET to be optimal. Furthermore, the use of Aspect.NET imposes some limitations on the target implementation. Most importantly, using *around interceptions*, Aspect.NET only supports intercepting static methods. This is unacceptable in many applications as many methods are created as instance methods. For example, the implementation of a generator for the C5 collection library would not be possible using Aspect.NET, as the implementation of C5 relies heavily on the use of interfaces, class interitance and polymorphism, which cannot be implemented using static methods.

Throughout the rest of this report we will focus on implementing an aspect weaver that directly addresses the shortcomings of Aspect.NET. Most importantly, we consider Aspect.NET's lack of support for making *around interceptions* of instance methods a crucial drawback. Around interception of instance methods should thus be supported by our aspect weaver. An important issue in this regard is to avoid any runtime performance penalties - the runtime performance should be directly comparable to an implementation coded by hand. The aspect weaver should be able to intercept methods, properties, class constructors and instance constructors as these are all required for implementing a generator for C5.

Another shortcoming of Aspect.NET is the lack of support for *introductions*. *Introductions* are important for expressing various types of interests. Our aspect weaver should thus support introducing various constructs, such as classes, methods, fields, etc.

At this time it is difficult to define any further requirements for our aspect weaver, as it requires more in-depth knowledge of the problems encountered when implementing an aspect weaver. In the next chapter we will thus analyze the various facets of the problem at hand and continously determine more detailed requirements for the aspect weaver. At the end of the next chapter all of the requirements will be settled.

Throughout the rest of this report we will refer to our aspect weaver as *YIIHAW*, which is a recursive abbreviation which stands for *YIIHAW Is an Intelligent and High-performing Aspect Weaver*.

Chapter 3

Problem analysis

During this chapter we will investigate the main issues involved in implementing a high-performing aspect weaver. The next section briefly describes the inner workings of Aspect.NET, evaluating the applicability of the techniques used herein. The subseeding sections analyzes some of the key concerns of aspect weaving and defines more detailed requirements for our own aspect weaver. Throughout this chapter we will assume that the reader is familiar with the *Common Intermediate Language* (CIL) of .NET. Consult chapter 4 for details about CIL.

3.1 Inside Aspect.NET

As shown in figure 2.8 Aspect.NET had a runtime overhead between 4 and 14%. To learn how Aspect.NET achieves this good result it is interesting to take a deeper look at the inner workings of the weaver. Aspect.NET is only available as a compiled program and it is therefore not possible to get the understanding from the source code[1]. What we will do instead is to try some of the different features of the weaver and then take a look at the CIL code that is generated.

3.1.1 Intercepting a method call

The possible interception semantics of Aspect.NET are *before call*, *after call* and *instead* (*around call*). In all three cases the implementation is very simple: The weaver transforms, not the class that contains the target method, but the methods that call the target method. For *before* and *after* interceptions a call to the advice method is placed right before or right after the call to the target method, as can be seen in figure 3.1(a) and 3.1(b). In the case of an *instead interception* the call to the target method is removed and a call to the advice method is inserted, see figure 3.1(c).

So the mechanism used is both very simple and powerful. However, as can be seen later in this section, this solution has some problems when the advice becomes a bit complicated. As Aspect.NET uses *call interceptions* this requires that the weaver has access to the locations from where the method is called. As new calls can come from external assemblies, i.e. assemblies not known at weave-time, this makes it impossible to guarantee that the advice will be called in all cases. Also, after the weaving there is still a dependency on the advice assembly as the weaver only inserts calls to the advice into the target assembly.

[1]We have been in e-mail contact with the developers of Aspect.NET and amongst other we asked if we could build this project as an extension to Aspect.NET or if they could give us any documentation regarding the design of the weaver. Both requests were unfulfilled[21].

```
a) "Before" advice weaved in before the call to the target method
  call       void [TestAspects]TestAspects.TestAspects::Aspect()
  call       void TestClass.Program::Target()

b) "After" advice weaved in after the call to the target method
  call       void TestClass.Program::Target()
  call       void [TestAspects]TestAspects.TestAspects::Aspect()

c) "Instead" advice weaved in instead of the call to the target method
  call       void [TestAspects]TestAspects.TestAspects::Aspect()
```

Figure 3.1: Aspect.NET interceptions.

3.1.2 Advice return type

For advice methods it might not always be appropriate that the method is able to return something. In Aspect.NET it is always possible to let an advice method return something. In these cases, the weaver does not perform any checking on the return type or what happens to the returned value, but leaves all typechecking to the user. The weaver just inserts the call as shown in section 3.1.1. For *before interception* this means that if the advice method returns anything, it is up to the programmer to make sure that something is done with this return value (which is not possible in a normal high-level language, when the advice is inserted the way it is). For *after interception* it would make sense to let the advice method use the value returned from the target method and possibly overwrite this value. However, in Aspect.NET the advice cannot handle the returned value from the target, which means that any value returned from the target method would be left on the stack, thereby generating the same problem as for *before interception*.

When it comes to *instead* it should of course be possible for the advice to return something, if the method it replaces returns something. This is clearly possible with Aspect.NET, but there is no check whether the return type is the same for the two methods. Furthermore, it is also possible to use an advice method with return type *void* for replacing a method that returns something. This can be seen in figure 3.2.

```
  //The original method calls method Target, which returns an int.
  //The returned value is popped from the stack
  call       int32 TestClass.Program::Target()
  pop       //removes the top item from the stack


  //Here the target of the method call has been changed to the Aspect() method
  //The Aspect() method does not return anything.
  //The following pop will try to remove the returned value from the stack.
  //This program cannot run. A System.InvalidProgramException is thrown.
  call       void [TestAspects]TestAspects.TestAspects::Aspect()
  pop       //Removes the top item from the stack - but there is none!
```

Figure 3.2: Aspect.NET return types.

3.1.3 Using arguments

The last feature of Aspect.NET that we will look into, is the possibility of using an advice method that takes arguments. The only way an advice method can access arguments in Aspect.NET is

by using the same arguments as the target method. When creating the advice one can specify whether it should use none of the arguments, all of the arguments or specific arguments from the target method. There is however no typechecking on the argument types.

Using *before* and *after*, Aspect.NET copies the opcodes for loading the arguments onto the stack and inserts them before the call to the advice method, as can be seen in figure 3.3. However, Aspect.NET's copying is incorrect in more complex cases. An example of this is shown in figure 3.4. In this example the instructions that load the locals (which are used as arguments when invoking *ArgMethod*) are copied, but not the instructions that initially stores something into the locals. The loaded locals do therefore not contain the right values.

Sometimes the weaver is not able to perform the weaving, especially if calls to methods are inserted directly as arguments (like *Foo(Bar())*). In these cases an exception is thrown which does not indicate what the problem is. So it seems that the weaver is very primitive when it comes to deciding on the amount of instructions to copy, which is not good enough for all cases. Furthermore, one has to be aware that when the argument is a reference type the same object is used in both the advice and target method, but when it is a value type they get two different copies.

```
//Here a before advice is weaved in and the two instructions that load the
    arguments onto the stack are copied in before the call to the advice method.
ldc.i4.2
ldstr       "teststring"
call        void [TestAspects]TestAspects.TestAspects::beforeArgAspect(int32,
    string)

//The original method starts here, which loads the two arguments and then calls
    ArgMethod()
ldc.i4.2
ldstr       "teststring"
call        void TestClass.Program::ArgMethod(int32, string)
ret
```

Figure 3.3: Aspect.NET passing simple arguments

When using *instead* the semantic is more clear, as the original method call is just changed - there is no need to copy any instructions. However, if it has been specified that only parts of the arguments should be passed to the advice method, then it is not certain that all the unnecessary instruction will be deleted, which might again result in Aspect.NET constructing an unverifiable program.

3.1.4 The lessons learned

Starting with the goal of figuring out why the output of Aspect.NET had such good results in the performance tests, we now believe that the insertion of calls to the advice method is the main reason. The small overhead that might be incurred by performing an extra method call is not very significant in the tests that we have performed. Furthermore, we have discovered that the solution used by Aspect.NET might be fast, but it also has a lot of drawbacks: The weaver is to simple when it comes to handling method arguments and return types. We believe that a different approach is needed when applying aspects in order to avoid the problems encountered when using Aspect.NET. Starting with the next section we will analyze various facets related to implementating an efficient aspect weaver that avoids many of the shortcomings of Aspect.NET.

```
//This is the high level language code of the target method
Random r = new Random();
int b = r.Next();
int a = 2;
ArgMethod(a+b, "teststring"); //This call is intercepted by Aspect.NET
```

```
//The following is the full method in CIL after the weaving. It is split up to
   show the orginal parts and the weaved part.

//The method has three local variables: A Random object and two ints.
.locals init (class [mscorlib]System.Random r,
         int32 b,
         int32 a)

//The original method starts here. The Random object is instantiated and is
   stored in a local variable.
newobj     instance void [mscorlib]System.Random::.ctor()
stloc.0

//The instructions inserted by the weaver start here. The first four of the
   instructions are copied from the original method by the weaver. They start
   by loading two local variables onto the stack. However, none of these two
   locals has been initialized yet, as the instructions that store values in
   them have not been copied by the weaver. This means that the code is not
   verifiable.
ldloc.2
ldloc.1
add
ldstr      "teststring"
call       void [TestAspects]TestAspects.TestAspects::beforeArgAspect(int32,
   string)

//And here is the rest of the orginal method.
ldloc.0
callvirt   instance int32 [mscorlib]System.Random::Next()
stloc.1
ldc.i4.2
stloc.2
ldloc.2
ldloc.1
add
ldstr      "teststring"
call       void TestClass.Program::ArgMethod(int32, string)
ret
```

Figure 3.4: Aspect.NET incorrectly passing composite arguments. The copied instructions loads the value of locals which has not been initialized yet.

3.2 Binding mode

Having looked at Aspect.NET, we now focus on how our own aspect weaver can be implemented. This section briefly discuss the types of weavers that exist and how they apply the aspects.

Basically two approaches exist when it comes to weaving aspects: Static weaving and dynamic weaving (also referred to as runtime weaving in some litterature).

Static weaving analyzes the targets and aspects and applies the aspects immediately as specified via the pointcuts. Having performed the weaving, the target assembly can be used immediately without any further processing. All aspects are applied at once to the target assembly. This is the technique used in Aspect.NET, AspectDNG and NKalore.

Dynamic weaving uses a different approach: Targets and aspects are written in a manner similar to that of static weaving, but the aspects are not applied until just before load-time. The aspect weaver inserts code into the target that applies the aspects on a Just-In-Time basis, i.e. the aspects are applied before the specific target is loaded into memory. This is a more flexible approach than static weaving, as it allows specifying pointcuts that can be varied between each program run or even while the program is running. This is useful if your aspects should be applied according to some events that are not known until runtime. However, this flexibility comes at a cost: As the weaving is performed runtime, it is naturally going to result in a runtime overhead compared to static weaving [7]. The significance of the overhead depends on the targets, the aspects and on how the actual weaving is performed. For some applications the performance overhead can be neglected.

A hybrid of the two techniques can actually be defined: Recall from section 2.1.1 that a static aspect weaver can use dynamic pointcuts, i.e. pointcuts that are determined at runtime. Such aspect weavers shares some features with dynamic weavers.

We believe that the use of AOP should be as efficient as possible and be directly comparable to that of an implementation coded by hand. The "Generation of specialized collection libraries" project [3] is a good example of a case where an aspect weaver generating efficient code is needed. Dynamic weaving will not be suitable for ensuring this requirement, as it introduces runtime overhead - no matter how efficiently the weaver is implemented, dynamic weaving will always require performing some operations at runtime. This means that runtime overhead cannot be avoided using this approach. Focusing on the runtime performance, static weaving thus seems to be preferable as it allows for an implementation where no (or very little) runtime overhead is incurred. We believe that the flexibility that is lost when choosing a static approach is neglectable in most situations as one would rarely need the kind of dynamic behaviour that dynamic weaving offers in performance-critical applications anyway. For instance, the "Generation of specialized collection libraries" project would have no use for dynamic weaving, as all requirements can be determined statically. For these reasons, we choose to implement a static aspect weaver.

A static aspect weaver that supports dynamic pointcuts offers the best of the two approaches: The user can decide whether he wants to apply only static advice or if he is willing to accept the runtime overhead of dynamic advice and use this instead. However, support for dynamic pointcuts makes the implementation of the aspect weaver much more complex as it requires that two different types of weaving are handled. As limited time is available for this project, we choose not to support dynamic pointcuts in our aspect weaver.

3.3 Applying aspects

The previous section stated that we consider static weaving to be the most suitable approach for implementing an efficient aspect weaver. With that in mind the following section looks deeper into how such a weaver can actually be implemented. Various techniques can be used for this purpose. Throughout this section we will describe the techniques we find the most interesting and analyze their applicability.

3.3.1 Source code weaving

Applying aspects directly to the source code is an obvious approach when handling aspects: By writing the targets and aspects in separate source files the aspect weaver can merge the files together, constructing one single entity. The aspect weaver should of course merge the files on the basis of some pointcuts defined by the user. This is illustrated in figure 3.5.

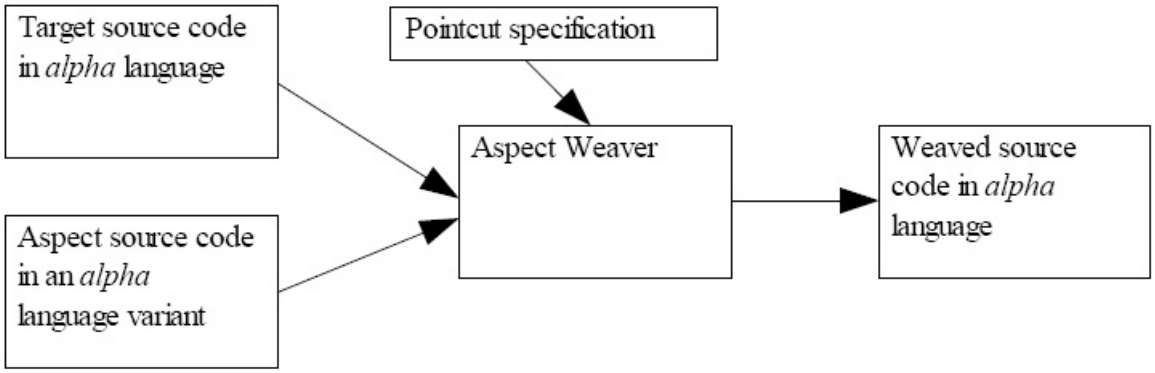


Figure 3.5: Source code weaving. Source code as input and output.

Working at a source code level allows the user to immediately see the changes applied, as the output is based on the same high-level language as the target code. This means that it is possible for the user to directly verify that the output is as expected and compile the code himself, using whatever compiler settings he prefer. Unlike most of the weavers presented in the previous chapter (which weaved at the assembly-level), it is thus fairly easy to track the aspects applied to the target. Furthermore, as the user is able to compile the output himself, it is even possible to debug the generated code using a standard debugger. This is not possible using any of the other weavers, as they directly manipulate the binary assembly, causing inconsistencies between the assembly definition and the debug file (PDB file).

Preprocessing

A variant of source code weaving can be defined: Preprocessing weaving. It uses the same principle as that of source code weaving, but outputs a compiled assembly instead of source code. This can be done by applying the aspects to the target code, which are then handed over to the compiler. The intermediate source code produced by the weaver is only used by the compiler and is thus not subject to further manipulation by the user. This principle is used in AspectC++ [12], where one can write aspects in the AspectC++ language. Using the AspectC++ weaver, which works both as a weaver and a translator of the AspectC++ language, a weaved C++ program is output. NKalore uses a similar approach.

CodeDom

As a specific solution for implementing source code (and preprocessing) weaving the *CodeDom* classes in .NET are an interesting case. The *System.CodeDom* namespace in the .NET framework contains classes which can be used to represent a program as an abstract syntax tree (AST). It is possible to traverse this AST by the nodes and leaves, which represents the different constructs in the program. An aspect weaver could alter the program either by changing the original AST or by introducing new code, by adding new nodes or leaves. When done weaving it is possible to either compile the program directly from the AST structure (useful for implementing a preprocessing weaver) or use a code generator to generate source code in the preferred language. CodeDom thus seems as a viable solution for implementing a source code or preprocessing aspect weaver.

The main drawback of using CodeDom is that even though there is an interface for it, there are no parsers to translate from source code to the CodeDom AST. As it does not make much sense to require the user to write his programs directly as a CodeDom AST (which is very cumbersome), it would be necessary to initially create a parser for the language of choice, as no such parsers exist at the time of writing [3]. Furthermore, not all constructs specified in the Common Language Specificaton are supported[19], which might turn out to be too restrictive when implementing an aspect weaver.

Is source code weaving the right choice?

Using source code (and preprocessing) weaving obviously requires that the source code for the target is available. In most cases this is not a problem, but for some applications this is unacceptable. Furthermore, creating a source code weaver also means that the weaver will be bound to one particular .NET language, whereas a lower-level weaver can be used for the whole family of .NET languages (as most of the weavers presented in the previous chapter). We consider source code weaving to be too restrictive in this regard.

Using source code weaving also means that there are no guarantees that typechecking has been performed on the code by a compiler prior to weaving the aspects. This means that the weaver is responsible for performing this typechecking in order to make sure that the aspects are valid in the new context. This check will be very cumbersome and errorprone to implement, as there will be no restrictions on the input from the user.

Another main problem with source code weaving, as we see it, is that it simply does not solve the key objective addressed by AOP: How to handle cross-cutting concerns efficiently. Injecting aspects into the target source code still produces *code scattering* and *code tangling* - it is just being introduced by an external program instead of by the user. Once the weaving has taken place the source code can be manipulated by the user in any way he likes, which means that he need to consider all of the concerns that has been introduced when working with the generated code. This is a problem, as all the cross-cutting concerns that was initially factored out in order to avoid code repetition are now included in the source code. This means that they cannot be said to have been encapsulated into an independent entity - they are rather defined once and are then applied as an advanced "search-and-replace" feature that does not respect the "separation of concerns" principle. Preprocessing weaving does not have the same problem, as it produces binary code instead of source code, effectively preventing the user from making any further modifications to the output. This means that a clearer separation of the targets and aspects are achieved. However, we still believe that the requirements of source code access and the strong binding to one specific .NET language are simply too restrictive, as they prevent the user from

taking an arbitrary assembly or source code file and apply whatever aspects he wants. We do therefore not consider source code or preprocessing weaving to be suitable solutions.

3.3.2 Direct advice invocation

In order to avoid the problems mentioned in the previous section, it makes sense to focus on a technique that can be used for working at the assembly-level, as this allows for a language independent approach. An obvious idea is to use the same approach as found in AspectDNG and Aspect.NET when handling interceptions[2]: A method call is inserted into the target methods at appropiate locations (which depends on the pointcut) that directly invokes the advice methods or, alternatively, invokes a version of the advice methods that has been modified slightly by the aspect weaver in order to fit into the new context. This concept is shown in figure 3.6: A target method, *SomeMethod()*, is defined which returns an integer. Furthermore, an advice method, *Advice()*, is defined which prints a message to the console, invokes the original method (Proceed) and returns the integer 7. Using the direct advice invocation approach, the aspect weaver creates two new methods: One for the first part of the advice (*ModifiedAdviceFirstPart()*) and one for the second part (*ModifiedAdviceSecondPart()*). These advice methods can either be inserted directly into the target assembly (as in AspectDNG) or simply be referred to from the target assembly via an assembly reference. The only modification that needs to be performed on the target method is that it should call the two new advice methods. This modification is fairly simple, as one only needs to change a few of the instructions in the target method. Most of the instructions can be ignored.

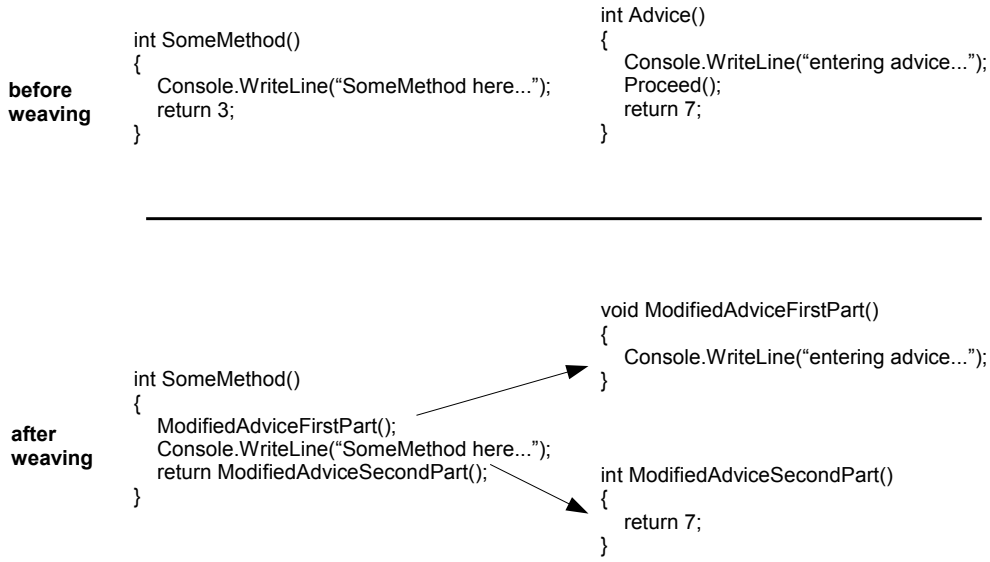


Figure 3.6: Direct advice invocation (using *around body interception*).

[2]Introductions are not important for this discussion as there is no alternative to inserting them directly into the target assembly. Likewise for typestructure modifications: There is no way to make a typestructure modification without actually going into the target assembly and modify the target classes. Thus, the only important feature when discussing assembly-level aspect weaving are interceptions.

An alternative solution to modifying the body of the target method is to modify the actual calls to the target methods instead, making them refer to the new advice methods. Using this approach the body of the target method does not need to be modified at all. This implies that *around call interception* (see section 2.1.2) is used instead of *around body interception*. Aspect.NET uses this approach.

As both targets and advice are compiled prior to invoking the aspect weaver, typechecking is somewhat simplified compared to source code weaving, as using compiled assemblies guarentees that they are verifiable. Basic checks on advice return type and advice arguments still need to be performed though, such as ensuring that the return type of an advice method is compatible with the method that it intercepts. All these checks should of course be performed statically in order to avoid runtime overhead.

Is direct advice invocation the right choice?

Although this approach might at first seem very useful, it does have some problems. First of all, directly invoking the advice methods means that these advice methods must either be inserted into the target assembly or be placed in a separate assembly that is then referred to from the target assembly. We believe that the appearance of advice methods within the context of the target assembly is a rather messy approach, as it clutters up the target assembly: It only makes sense to invoke the advice methods from within one of the intercepted target methods, but nothing prevents the user of the target assembly from invoking these methods himself outside their intended context. This might lead to strange behaviour. We believe that the use of interceptions should not introduce any unwanted constructs within the target assembly - it should be impossible to see if interceptions has been applied to the assembly or not. This cannot be implemented using direct advice invocation.

A second problem is the small overhead incurred when inserting the extra calls to the advice methods. Method calls will always take a small amount of time to execute, even though this amount may in most cases be so small that it is not significant (as the results of Aspect.NET also showed in the previous chapter). However, we believe that the use of interceptions should not result in any runtime overhead at all. For these reasons, we do not consider the use of direct advice invocation to be a proper solution.

3.3.3 Inlining advice

The previous discussion created the need for a technique that operates at the assembly-level, but does not introduce any unwanted constructs in the target assembly. One such approach is to inline the advice within the target method. This concept is illustrated in figure 3.7.

As can be seen, the aspect weaver should merge the two methods, creating one combined method that substitutes the intercepted target (thus performing an *around body* interception). This means that the actual weaving is done on the basis of the IL-instructions found in the target and advice assemblies.

As shown in figure 3.7, using inlining usually requires some means of modifying the advice and target methods, making them fit into the new context. Some IL-instructions will be redundant (such as the *return* instruction of *SomeMethod()* in the example shown in figure 3.7) and should be removed. Other instructions need to be modified in order to be compatible with the new context. The weaver should be able to make these transformations and produce a valid .NET program, containing the combined functionality of the target and the advice methods.

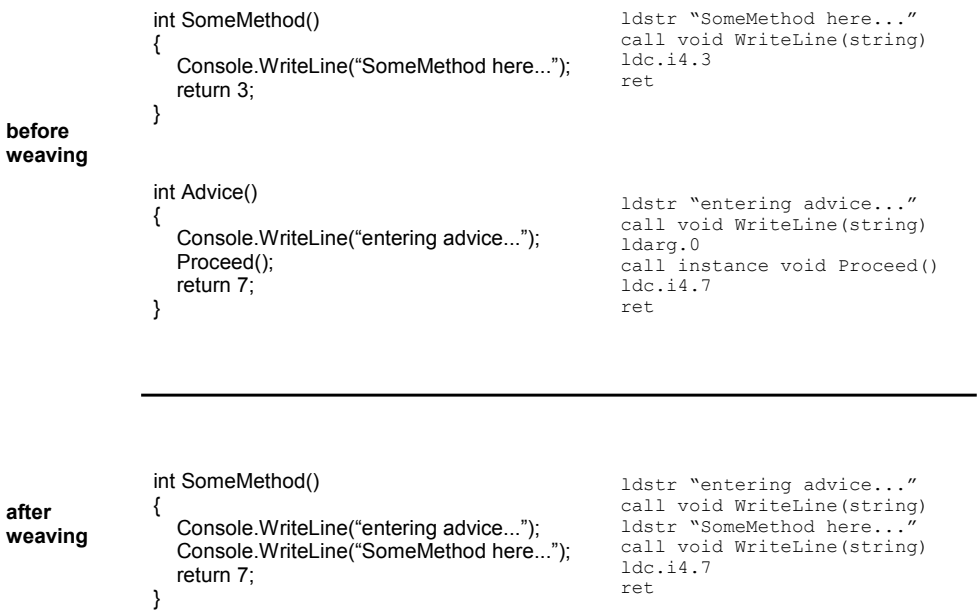


Figure 3.7: Inlining the advice within the target method. Shown to the right is the IL assembler representation of the source code.

Is advice inlining the right choice?

An inlining approach is obviously more cumbersome to implement than using direct advice invocation. However, we believe that this technique allows us to achieve the fine-grained control needed for implementing an efficient aspect weaver: Being able to control the generated output down to a single IL-instruction allows for great optimization when weaving the advice and target, as it makes it possible to handpick each instruction that should be woven into the target.

Another main advantage obtained from using an inlining approach, is that unnecessary constructs can be avoided altogether in the output assembly: There is no need to insert any auxiliary methods into the output assembly, which means that the woven methods will be completely self-contained. We consider the introduction of these unneeded constructs in the output assembly to be one of the biggest problems in many of the other aspect weavers, as it breaks the program-structure defined by the user. We believe, that the aspect weaver should respect the structure defined by the user and apply the advice in a non-interruptive way.

For these reasons, we consider the use of advice inlining to be the most suitable approach for implementing a high-performing aspect weaver. Throughout the rest of this chapter we will further analyze how such an approach can be implemented in practice and identify some of the key objectives in this regard.

3.4 Implementing aspects

Having defined which technique should be used for implementing the aspect weaver, the following section discusses some preliminary ideas of what the aspect language should look like, i.e. how the actual aspects should be implemented by the user and how the user should instruct the weaver to do as desired.

An important facet of an aspect weaver is that it should support the features that the user wants while keeping the implementation of the aspects as simple as possible. The implementation of the aspects should be as transparent and direct as possible: The user should not need to learn a whole new programming model or be forced to implement things in a different manner than he is used to. The implementation of the aspects should be comparable to the implementation of any ordinary program - no special actions or precautions should be made when writing the code. In order to support this requirement, care has to be taken when designing the aspect weaver.

3.4.1 Introductions

As mentioned in section 2.4, YIIHAW should support introducing new constructs, such as methods, fields, delegates, etc. into the target assembly. No special syntax should be used when defining these constructs - all constructs should be defined as in any ordinary .NET program, e.g. a delegate that you want to insert could be defined like this:

```
public delegate int MyDelegate(string s, int i);
```

Having defined this delegate, it should be targetable by a pointcut description (discussed in section 3.5). The aspect weaver should support introducing any type of construct, i.e. the definition of the constructs should not be restricted to certain types or notations or require the use of special annotations within the code.

Inserting new constructs should trigger the weaver to typecheck all references within those constructs, as they may themselves refer to other constructs that do not exist in the target assembly at the time of weaving. Consider the following code sample:

```
public int NewMethod()
{
  ...
  AnotherMethod(); // invoke AnotherMethod - the aspect weaver should check that
      this method exist within the target assembly
  ...
}
```

In this case, the aspect weaver should check that *AnotherMethod()* already exist within the target assembly or that it will be inserted at some point (i.e. that the user has instructed the weaver to insert it via a separate pointcut statement). If none of these requirements are met, the weaving should be cancelled as the call to *AnotherMethod()* in that case makes no sense in the new context (and would result in a runtime error when executing the program). A similar check has to be made for any other construct (field, property, delegate, etc.) that is referred to from within the constructs that are introduced into the target assembly. Of the aspect weavers examined in the previous chapter, only NKalore make this kind of typechecking.

3.4.2 Interceptions

The process of intercepting methods is probably the most important feature of AOP, as it is what basically allows handling the cross-cutting concerns, which is the key motivator for using AOP in the first place. The aspect weaver should allow the user to achieve a fine-grained control of the interception process while still hiding as many of the problems of inlining IL-instructions as possible. For instance, the user should not need to be concerned about how a local variable within the advice gets translated into the target assembly or how the advice method is aligned within the method that is intercepted. Such details should be isolated as much as possible from the user and be handled by the weaver.

Types of interceptions

Section 2.1.2 stated that three kind of interceptions can be defined: *before*, *after* and *around*. These can be further divided into *call* and *body*. Recall that a *call interception* only modifies calls to the target method - it does not modify the target body itself. We believe that *call interceptions* are not as efficient as *body interceptions*, as *call interceptions* in most cases imply that at least two method calls has to be performed instead of one: One to invoke the advice and one to invoke the original target. Furthermore, using *call interceptions* it will in some cases not be possible to intercept all references to a specific target as these references cannot always be determined statically: The target might be invoked from another assembly after the point of weaving, in which case the reference will not have been intercepted by the weaver. Using *body interceptions* this issue is avoided, as the target is always modified. This means that all references to the intercepted target will automatically invoke the modified target after the point of weaving. For these reasons, we choose to implement interceptions using an *around body* approach.

From a logical perspective, *before* and *after* are redundant as they can be simulated using *around*. However, they are often supported to improve the runtime performance. We believe that the use of advice inlining makes it possible to avoid the performance penalty of *around interceptions*, as it allows a low-level control of both targets and advice. This low-level control can be used to handle the weaving process directly at the instruction-level, which means that we are not restricted to using higher-level methods, such as reflection, method calls or generation of proxies. For this matter, we will not implement support for making *before* and *after* interceptions, as we consider them to be redundant.

Handling the join point

Implementing advice often requires access to information related to the method that is being intercepted. Most weavers support some way of getting various information about the current join point, such as the name of the method, it's return type, the access specifier, etc. Having access to this kind of information is essential if one must implement a logging advice, which is one of the most common cross-cutting concerns of AOP. If the user cannot require information about the current join point, there is little use for a logging advice in the first place, as one cannot log which method was entered. To accommodate these requirements YIIHAW should provide access to the following join point information:

- Method signature (name, parameters and return type).
- If the method uses any generic type parameters, these should be available as well.
- Access specifier of the method (public, private, etc.).

- Invocation kind (static or instance).

Having access to this kind of information allows the user to write pseudo-advice such as this this:

```
public void Advice()
{
  Logger.Write("entering method: " + MethodSignature);
  ... invoke the orginal method ...
}
```

Of the weavers presented in the previous chapter, only AspectDNG does not support this feature.

Interacting with the target method

The most common use of interceptions is to add additional code to an existing method. This means that at some point in the advice the original target method will be invoked by the user. Most aspect weavers support invoking the original method, usually via a method called *Proceed()*. Calling this method allows the user to invoke the original method and acquire whatever value was returned. This value can then be subject to further manipulation or can be returned directly. The small pseudo-code sample shown below illustrates the typical usage of *Proceed()*.

```
public class X
{
  public int TargetMethod()
  {
    return 3;
  }

  public int Advice()
  {
    ... some advice code ...
    int result = Proceed(); // invoke target method and store its result
    return result; // return the same value as the target method returned
  }
}
```

Aspect.NET is the only weaver that does not directly support the use of *Proceed()*. Instead the user has to invoke the original method directly from within the advice. We believe this approach makes it difficult to make generic advice as one has to know which method is being intercepted and use that knowledge to invoke the correct method. Using the other weavers, one can just invoke *Proceed()* and let the weaver make the decision on which method to call. Furthermore, invoking the target method directly would not be appropriate for anything but static methods (which is the only type of methods that Aspect.NET can intercept), as invoking an instance target method would require a reference to the correct object instance within the advice in order to make the call. This is shown in the small sample below.

```
public class X
{
  public int TargetMethod()
  {
    return 3;
  }

  public int Advice()
```

```
  {
    ... some advice code ...
    X original_object = Target; // get a reference to the object containing the
        target method
    int result = original_object.TargetMethod(); // invoke the target method using
         the obtained reference and store the value returned
    return result; // return the same value as the target method returned
  }
}
```

This advice is obviously more errorprone to write than the one shown previously, as the user is responsible for invoking the correct method. We believe that the implementation of the advice should be as simple as possible for the user. Providing a *Proceed()* method makes sense, as it simplifies the users interaction with the target method and allows for more generic types of interceptions. YIIHAW should thus support such an approach.

Accessing the arguments of the target methods is another important issue. The weaver should support access to these arguments, as it is often necessary to use them within the advice. Except for AspectDNG, all of the weavers presented in the previous chapter support accessing the arguments of the target method. Basically, two approaches are common for handling this:

1. *Provide constructs for fetching the arguments:* Create a method or property that can be used to retrieve the arguments. For instance, create a method, *GetArguments()* that returns an array of objects. The user can then typecast the arguments to the correct type and use them within the advice.

2. *Use shadow methods:* Create the advice method with the same signature as the target method and use the arguments directly within the advice. The aspect weaver is then responsible for translating references from *advice.argumentX* to *target.argumentX*.

The former approach is obviously less typesafe than the latter: By operating on an array of objects the user is responsible for typecasting the arguments to the correct type. This typecast cannot be checked by the compiler. Instead, the weaver should try to typecheck the casts and see if they make sense by comparing them to the actual types of the arguments in the target method. Even worse, *ref* and *out* arguments cannot be supported using this approach, as there is no way to store these type of references in an array. Furthermore, all value types will need to be boxed when stored in an array - this obviously leads to runtime overhead (albeit very small for most cases). We also consider this approach to be much different from the usual way of handling arguments, as it requires the user to apply a whole different way of thinking when it comes to dealing with method arguments.

We believe that using shadow methods is a better approach, as it ensures type safety and an easier programming model. Furthermore, using this approach actually simplifies the weaver as well, as it does not need to typecheck the usage of the arguments - all of this is handled by the compiler.

Weaving the target and the advice

The advice samples shown so far in this chapter have all been somewhat simplified in order to focus on the details that are interesting for the discussion and leave out the irrelevant ones. As mentioned earlier in this section, we require that the implementation of the advice should hide as many of the low-level details from the user as possible. Consider the sample advice shown below (this sample is equal to the one shown earlier).

```
public class X
{
  public int TargetMethod()
  {
    return 3;
  }

  public int Advice()
  {
    ... some advice code ...
    int result = Proceed(); // invoke target method and store its result
    return result; // return the same value as the target method returned
  }
}
```

This advice simply invokes the *Proceed()* method, stores the return value in a variable, *result*, and then returns the value of *result*. Notice that the return type of the target method, the advice method and the *Proceed()* method conveniently happens to be of the same type (*int*). This is because this sample has been simplified for the sake of discussion. Obviously, in general, methods do not just return *int* - they can return any kind of object (or void). The aspect weaver should of course be able to handle other types as well. A somewhat more "general" implementation syntax might look like this:

```
public class X
{
  public int TargetMethod()
  {
    return 3;
  }

  public T Advice()
  {
    ... some advice code ...
    T result = Proceed(); // invoke the target method and store the value returned
    return result; // return the same value as the target method returned
  }
}
```

Notice that the return type of the advice method and the *Proceed()* method have now been modified to return any type, *T*, instead of *int*. They have been "generalized" in order to support any type of return type of the target method (which is still *int* in this example). The reason behind this syntax is as follows: By making the advice and *Proceed()* method return any type, they can intercept any kind of target. The user does not need to worry about the return type of the target method when making this kind of interception. The return type could be *int*, *string*, *ICollection* or even *void* - for all cases the implementation looks the same. This makes sense as the user should be able to make an advice method that can intercept all methods, regardless of the return type. We believe that this kind of implementation offers the best solution for the user when it comes to hiding the details behind the interception. AspectDNG uses a similar approach for hiding the details of which type is returned from the advice.

When our weaver sees this kind of advice it should replace the return type of the advice method and the *Proceed()* method with the correct return type (which is the return type of the target method) when generating the CIL code. This ensures that no runtime casts need to be performed. So the weaver should translate the advice into a proper context that fits whatever method is being intercepted. This way, the user never needs to worry about how to write an advice method for at specific return type - this problem should be handled by the weaver. No

aspect weaver that we are aware of makes this kind of optimization.

Accessing the receiving object

A similar problem is how to handle the receiver of a target instance method (the object containing the method): Most aspect weavers support some way of getting a reference to the object that is being intercepted. This is useful if you want to call methods or access field or properties on the target object. As the target object and the advice object are completely independent at the time of declaration, there must be some way to get a reference to the original object from within the advice. This is usually supported via some method or property called *Target* or similar. The sample below shows how one can get a reference to the original object (pseudo-code):

```
public class X
{
  public int TargetMethod()
  {
    return 3;
  }

  public void SomeMethod()
  {
    ...
  }
}

public class Aspects
{
  public int Advice()
  {
    ... some advice code ...
    X original_object = Target; // get a reference to the object containing the
        target method
    original_object.SomeMethod(); // invoke a method on the original object
    ... some advice code ...
  }
}
```

The advice method uses a property called *Target*, which returns a reference (of type X in this sample) to the original object and invokes *SomeMethod()*. This type of operation is supported by most weavers and should be supported by our weaver as well, as such an implementation is often needed. However, when weaving this kind of advice the weaver should make some optimizations: First of all, as the weaving effectively merges the two methods there is no need to invoke a special property in the CIL code when acquiring a reference to the current object. When the weaving takes place this property invocation should thus be replaced by a direct reference to the object (the *this* reference). This is possible at weave-time as the two methods are effectively combined into one. Futhermore, the weaver should check that the enclosing type of the target method is actually of type X. If this is not the case, the weaving should be aborted. By making these optimizations during weave time no runtime overhead is added. No other aspect weaver that we are aware of uses such optimizations.

Note that although the problem described above does share some characteristics with the *Proceed* problem described previously, they cannot be solved in the same manner. When using *Proceed*, one often does not want to consider the actual type being returned. This was the key intention behind using generics for handling this problem: By specifying that the advice method returns T means that the weaver should infer the actual type being intercepted and replace T

with that type. This is not the case when referring to the receiver of the intercepted target method, as one should always know the concrete type of the receiver: It makes no sense to invoke a method or update a property on the receiver without specifying the actual type, as the methods and properties are not available on a generic type. The compiler would naturally reject such an attempt to use unavailable methods or properties.

Typechecking references

At last, the aspect weaver should typecheck all references within the advice body. In the advice, one might invoke other methods or other constructs, such as properties, fields, delegates, etc. All of these references should be checked by the weaver in order to determine if the reference "makes sense" in the new context. Consider the following example:

```
public class Aspects
{

  public void SomeMethod()
  {
    ...
  }

  public T Advice()
  {
    SomeMethod(); // invoke SomeMethod()
    return Proceed(); // invoke the original method (not shown here) and return it
  }
}
```

The advice invokes a method called *SomeMethod()*. The compiler has no trouble compiling this because *SomeMethod()* is declared in the same class as the advice method. However, when weaving the advice into the target method the aspect weaver should check that *SomeMethod()* exists within the target class. This check is complicated by the fact that *SomeMethod()* might itself be subject for insertion: The user might want to insert this method within the target class (specified via pointcut) and then invoke it. This should be accepted by the weaver. If the method is not defined within the target class and is not subject for insertion into the target class, the aspect weaver should abort the weaving process and show an error.

3.4.3 Typestructure modifications

YIIHAW should support making modifications of the typestructure as well. Two types of modifications should be supported:

- Changing the inheritance tree.
- Implementing one or more interfaces.

Changing the inheritance tree allows the user to change which type a target type should extend. Similarily, the weaver should support changing target types so that they implement one or more interfaces. This is also useful when generating programs via AOP. When changing the superclass or implementing an interface, the weaver should check that interface and abstact methods are implemented by the target class.

Of the aspect weavers described in the previous chapter, only AspectDNG supports changing the typestructure of the target classes.

3.5 Pointcut specification

The pointcut language is one of the most important parts of an aspect weaver. It is the pointcut language that determines which constructs can be targeted and how the aspects can be applied. Thus, the pointcut language serves as the users primary means of interacting with the weaver. In this section we will discuss some of the key issues involved in designing a pointcut language. At this point focus will be kept on some of the higher-level issues; details will be deferred to later.

3.5.1 Writing the pointcuts

Before designing the pointcut language one must decide where the pointcuts should be written. Generally, two approaces are used in the aspect weavers presented previously: Annotate the aspects or targets using .NET attributes or define the pointcuts in a separate file.

Using .NET attributes, the user can directly annotate either the target or the aspects with a pointcut description. AspectDNG and Rapier LOOM supports defining the attributes in the target code. This approach obviously requires access to the source code of the target. Aspect.NET supports defining the attributes directly on the aspects, i.e. inside the aspects assembly.

Another approach is to define the pointcuts in a separate file. This allows for a total separation of the aspects, the target and the definition of where to apply the aspects. This approach is supported by AspectDNG and Aspect.NET.

The advantage of using the annotation approach is that it forces the user to clearly define where the aspects apply within the source code for the target or aspects. This obviously makes the code somewhat easier to understand, as it is explictly stated which aspects apply where. On the other hand, using a separate file for the definition of the pointcuts require that the user must himself estimate where the aspects apply by looking at the pointcut file and possibly the source code. However, we believe that the annotation approach does have some drawbacks that make it less suitable than using a separate file: Defining the pointcuts directly on the target (as it is done in AspectDNG and Rapier LOOM) requires that the source code for the target is available. Acquiring the source code is not always possible. Using this approach thus demands that the target assembly must be explicitly aware of the aspects that is going to be applied to it, hereby breaking the principle of *obliviousness*. This means that you cannot take some arbitrary assembly and extend it with whatever functionality you want, as this assembly will in most cases not support the use of aspects (or at least not the aspects that you want). We consider this to be huge drawback.

Annotating the aspects assembly is a better approach, as it does not enforce any requirements on the target assembly. Of course, annotating the aspects assembly require that the source code for the aspects are available. However, this is in most cases not a problem, as the aspects are often created ad hoc on the basis of some target; very rarely would you use some arbitrary aspect assembly when applying aspects. However, we still believe that annotation of the aspect assembly has one major problem: Writing aspects in this manner requires that all pointcuts must be defined at the point of compilation. You cannot modify or extend the pointcuts once the aspects have been compiled. Using a separate pointcut file allows greater flexibility, as it makes it possible to implement and compile the aspects separatly and postpone the decision on where to apply the aspects. This approach is very useful if you need to define the pointcuts based on some events or settings that are not known at the time of implementation. The "Generation of specialized collection libraries" project [3] uses such an approach: All aspects

are implemented and compiled once and for all, while the pointcuts are generated on the basis of a configuration file. This reduces the complexity of the aspects dramatically, making the handling of dynamically created pointcuts much easier. For this reason, we believe that defining the pointcuts in a separate file is the most appropiate implementation.

3.5.2 The pointcut language

The pointcut language should be able to express all types of constructs that the user may wish to target while still being as simple and easy to understand as possible. This implies that the pointcut language should support a coarse-grained syntax that allows the user to match many constructs using one statement, while still supporting the use of a more fine-grained syntax when ever this is needed.

Generally, three types of elements should be supported by the pointcut language: Interceptions, introductions and typestructure modifications. At the least, the pointcut language should be able to support the following expressions:

- Intercepting methods.
- Introducing methods.
- Introducing fields.
- Introducing classes.
- Modifying the typestructure.

These expressions are the most commonly used and should thus be supported by our pointcut language.

3.5.3 Wildcards

All aspect weavers described in the previous chapter support the use of *wildcards*. Wildcards allow the user to match all constructs using a special syntax (most oftenly a '*'). This allows great flexibility when targeting many constructs using one statement, e.g. when implementing a "catch-all" logging advice. As the use of wildcards greatly reduces the amount of repetitive text that needs to be written, our pointcut language should be able to support these on all target constructs as well.

3.5.4 Complex expressions

Some aspect weavers support the use of complex expressions in the pointcut containing many conditions in one statement, such as:

```
(A and (B or C)) or D
```

This kind of syntax allows the user to achieve a very fine-grained control of what should be targeted by the pointcut. However, such expressions are most oftenly used when describing dynamic pointcuts, such as *cflows*, where they are useful for describing the scope of the construct that should be targeted. Complex expressions are seldom used for describing static pointcuts, as no scope needs to be defined for these types of pointcuts. As our aspect weaver will be purely static, we do not consider it relevant to support these kind of expressions.

3.6 Summary

The requirements for our aspect weaver are summarized below.

- The weaver should support static binding only, i.e. all constructs must be determinable statically (section 3.2)
- All advice should be inlined within the target methods (section 3.3.3)
- Only *around body interceptions* should be supported (section 3.4.2 - *Types of interceptions*)
- The weaver should typecheck all references in order to make sure that they are valid in the new context (section 3.4.2 - *Typechecking references*)
- The weaver should provide access to various join point information (section 3.4.2 - *Handling the join point*)
- Shadow methods should be used for accessing method arguments of the target methods (section 3.4.2 - *Interacting with the target method*)
- The weaved assemblies should be completely self-contained once the aspects have been applied, i.e. the generated assemblies should not be dependent on any external aspect assemblies or the presence of any aspect constructs (section 3.3.3)
- All pointcuts should be defined in a separate file (section 3.5.1)
- The pointcut language should support the use of wildcards where possible (section 3.5.3)
- The weaver should support interceptions, introductions and typestructure modifications (section 3.4)
- When modifying the typestructure, the weaver should check that all interface methods and abstract methods are implemented by the target class (section 3.4.3)
- The use of *Proceed* should be supported, as it simplifies the advice syntax (section 3.4.2 - *Interacting with the target method*)
- The aspect language should provide a convenient syntax for writing advice methods that can be used for implementing "catch-all" interceptions. In this regard, the weaver should map all returns types into their proper type (section 3.4.2 - *Weaving the target and the advice*)
- Accessing the intercepted objects should be supported by the aspect language (section 3.4.2 - *Accessing the receiving object*)

Chapter 4

Common Intermediate Language

Based on the choice of using advice inlining, which we decided on in the previous chapter, we will in this chapter take a look at the Common Intermediate Language, which is the intermediate language representation that all .NET languages are compiled into. Furthermore, we will also describe how the CIL code is executed through the Common Language Runtime.

4.1 What is CIL?

Instead of compiling programs directly to the assembler language of one specific computer architecture, the choice made when designing .NET was to use an architecture independent representation. This is called the Common Intermediate Language (CIL). Other than the portability between architectures some of the interesting facets of CIL is that it:

- opens up for easy collaboration between the languages in .NET (language interoperability).
- is a a human readable low-level language.
- is a stack based system.

These three facets will be further elaborated upon in the following sections. Some of the information and examples given in the following sections is based not on the actual definition of CIL, but on the way CIL can be written in the .NET 2.0 IL Assembler (*ilasm*), which is distributed along with the .NET framework and is the most commonly used compiler for CIL code.

4.1.1 Assembly, modules and metadata

An assembly is the main output of a compilation of one or more source code files which are bound together. When working with advice inlining, assemblies will be the input and output of the weaver. Looking at an assembly as the root of a hierarchical structure, it contains modules which are managed Portable Executeable (PE) files (normally .exe or .dll)[1], the modules contains types, and the types contains methods, events, fields and properties. Furthermore, the assembly and the modules contains a lot of metadata, which amongst other uniquely identifies the assembly and modules, states external dependencies (both resources and other assemblies) and decribes the types, methods, events, fields and properties in the modules. It is this metadata that makes the collaboration between languages easy, as the language independent description of the program structure opens up for access from all the languages in the .NET family. In relation

[1]The PE file format is an executable format specified by Microsoft. Its main strength is that it is portable between all 32-bit operating systems created by Microsoft.

to aspect weaving this metadata is very useful, as the information it contains make it easy to navigate the structure of the program and work with the dependencies of the parts that should be interweaved.

4.1.2 Instructions and operands

What is missing in the hierarchical structure described in the previous section are the instructions that a method contains. The CIL instruction set contains a total of 218 instructions [2] (in .NET version 2.0), and it is the possibility of working directly with these instructions that makes it possible to use advice inlining. Each instruction consists of an opcode which indicates the action that should be performed, and sometimes an operand that contains extra information for the opcode, such as what local variable to use, a specific string to load, or an offset address to jump to. Figure 4.1 shows examples of these.

Opcode	Operand	Action
`ldloc.0`		Loads the first local variable onto the stack. Here the opcode holds the information on what to load.
`stloc`	`a`	Store the top value from the stack into local variable *a*. Here the operand holds the information on where to store.
`ldstr`	`"Hello"`	Loads the specified string onto the stack.
`br.s`	`1234`	Branches with the offset given in the operand. The *.s* indicates that this is the short version of this opcode, which only takes an 1-byte operand.

Figure 4.1: Examples of different instructions.

The operand is in many cases a 4-byte value, but for each opcode that takes such an operand, there also exist a short form version which takes only a 1-byte operand, and in special cases (load of arguments, load and store of local variables, and load of constants) there are even versions of the opcodes which does not take any operand. An aspect weaver that focus on achieving as little runtime and space overhead as possible should of course always use the most optimized version of these opcodes, so that the program is kept as compact as possible.

4.1.3 Flow control

A special subset of the instruction set are the instructions that can change the linear flow of program execution. Except for the *ret* instruction, which indicates that the execution should return to the caller, all the flow control instructions take an operand which indicates the offset of the branch. These instructions are specially interesting when inlining advice, as the offsets might change when the advice and the target are weaved together. When using the IL Assembler (*ilasm*) it is possible, instead of specifying precice offsets as operands for the branch instructions, to label specific locations in the CIL code and then use these labels as the operands of the branch instructions. This is one of the things that makes the code more readable for humans, but it also make it a lot easier to program, as calculating the offsets can be avoided. Of course this can be used when the aspect weaver needs to change the operands of the branch instructions, if the weaver uses the IL assembler.

4.1.4 Datatypes

The primitive datatypes defined in CIL is another thing that opens up for language interoperability. All the .NET languages share the same primitive datatypes and thereby avoids any problems with differences in the types between the languages. In the case of the .NET framework, all the primitive datatypes are represented by classes in the *System* namespace and can be reprensented both by their full signature, like *System.Int32* or by their short forms like *int* when using IL Assembler.

CIL also contain special datatypes, called generics. These are abstract datatypes which works as templates for fully defined types, which will be given at instantiation time. In the metadata, a non-generic type and a generic type only differs by the list of generic parameters that the generic type holds. So to check if a type is generic it is necessary to check the parameter list in the metadata. Another important fact is that it is possible to put constraints on generic types, thereby specifying that the actual type given at instantiation should extend and/or implement the types that are used as constraints on the generic type.

To fully explain generic types and the special rules concerning them is not in the scope of this text, but the facts given here are the most important ones to be aware of if generics should be supported by our aspect weaver.

4.1.5 The stack

As mentioned in the example in figure 4.1 the load and store instructions pushes values onto the stack and pops values off the stack. CIL is entirely stack based, meaning that all computations performed on the data is done through the use of a stack. The stack is not working with a specific size for each entry, but with slots. Each slot has the size needed for the value that is stored in it. As speciale rules apply for the use of the stack, attention is also needed when creating an inlining aspect weaver. For example the stack always need to be empty when a method is returning (except for a possible return value) - a rule that the Aspect.NET weaver does not obey (as described in section 3.1.2). One of the metadata directives is *.maxstack*. The *.maxstack* directive is used for each method definition, and it contains an indication of the maximum size of the stack for the method in question. If the directive is not stated the default size is 8 slots. As the weaving of two methods can change the use of the stack, it might be necessary to change the *.maxstack* value after weaving. Figure 4.2 shows an example of this.

4.1.6 Exception handling

A special part of the metadata in an assembly is exception handling. Exception handling is controlled by the Common Language Runtime, guided by the exception handling metadata specified for each method. Using the IL Assembler the exception handling metadata can be specified in two ways: Either it can be specified in scope form, which means that the directives are inserted at the proper locations in the IL Assembler code where it encloses the scope of the handler (figure 4.3(a)), or it can be placed in a section after the IL-instructions, where it must use labels for indicating the scope (figure 4.3(b)).

```
//Target method                 //Target method CIL code              //Stack before
                                .maxstack 2
int Target()                    .locals init ([0] int32 a)
{                               ldc.i4.2                            []
 int a = 2;                     stloc.0                             [int]
 return a*2;                    ldloc.0                             []
}                               ldc.i4.2                            [int]
                                mul                                 [int,int]
                                ret                                 [int]



//Advice method                 //Advice method CIL code              //Stack before
                                .maxstack 2
void Advice()                   ldc.i4.2                            []
{                               call int32 Proceed()                [int]
 Foo(2, Proceed());             call void Foo(int, int)             [int, int]
}                               ret                                 []
```

```
//After weaving
//target method replaces Proceed() and the "ret" instruction is removed.


//The woven method                                                  //Stack before
.maxstack 3
.locals init ([0] int32 a)
ldc.i4.2                                                          []
//this is where Target() is inserted.
ldc.i4.2                                                          [int]
stloc.0                                                           [int, int]
ldloc.0                                                           [int]
ldc.i4.2                                                          [int, int]
mul                                                               [int, int, int]
//end of Target() insertion
call Void Foo(int, int)                                           [int, int]
ret                                                               []
```

Figure 4.2: After the weaving is performed, the maxstack value needs to be higher than it was in the target and the advice before the weaving. This is necessary as the advice puts something onto the stack right before it inserts the target method.

```
//C# code
void Foo()
{
    try
    {
        Fun(1,2);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

```
A - exception handling - in scope:

.method private hidebysig instance void  Foo() cil managed
{
  .try
  {
      ldc.i4.1
      ldc.i4.2
      call       void test.Class1::Fun(int32, int32)
      leave.s    IL_0017
  }  // end .try
  catch [mscorlib]System.Exception
  {
      stloc.0
      ldloc.0
      callvirt   instance string [mscorlib]System.Exception::get_Message()
      call       void [mscorlib]System.Console::WriteLine(string)
      leave.s    IL_0017
  }  // end handler
  ret
} // end of method Class1::Foo
```

```
B - exception handling - in its own section:

.method private hidebysig instance void  Foo() cil managed
{
  Label_A:  ldc.i4.1
            ldc.i4.2
            call       void test.Class1::Fun(int32, int32)
            leave.s    Label_C
  Label_B:  stloc.0
            ldloc.0
            callvirt   instance string [mscorlib]System.Exception::get_Message()
            call       void [mscorlib]System.Console::WriteLine(string)
            leave.s   Label_C
  Label_C:  ret

  // Exception count 1
  .try Label_A to Label_B catch [mscorlib]System.Exception handler Label_B to
      Label_C
} // end of method Class1::Foo
```

Figure 4.3: Two ways of specifying exception handling. A - in scope, B - in a separate section.

Chapter 5

Working with CIL

As can be seen in the previous chapter there are a lot of facets to be aware of when working with CIL. With the features that YIIHAW should support, it will be necessary to work with many parts of CIL and we will therefore in this chapter look into how this can be handled.

The reading and writing of assembly files and the manipulation of CIL code and assemblies in general is not a trivial task. Even though the *IL Disassembler (ildasm)* and the *IL Assembler (ilasm)* included in the .NET SDK can be helpful when it comes to reading and writing the assemblies, there are still a lot of things that need to be controlled in order to work with more than just simple assemblies. First of all there is the actual declaration of the assembly manifest and its metadata, which includes defining dependencies, permissions, resources, content and much more. Secondly, working with the IL-instructions demands a thorough knowledge of the opcodes and how to specify their operands. The calculation of offsets can in itself be a difficult task.

Instead of spending time on building our own tools to do the assembly and CIL manipulation, we have chosen to take a look at two of the most feature-rich tools available today.

5.1 Microsoft Phoenix

Microsoft Phoenix [18] is the code name for a software optimization and analysis framework, which is planned to be the foundation of all future compiler technologies from Microsoft for at least the next 15 years. The purpose of the framework is to give developers and researchers an environment in which compilers, programming tools, language tools and new languages can be built in an effective manner. The framework exposes an intermediate representation(IR) as the central building block. The IR is composed of both a high-level IR and a low-level IR, giving the possiblity of working on different abstraction levels. Through the Phoenix API it is possible to analyze and manipulate the IR and afterwards output the IR in different output formats.

In the current release of the Phoenix Research Development Kit (May 2006 release) readers are included for CIL and Microsoft portable executable (PE) binary image file. Likewise, the output writers can generate CIL, PE files and COFF object files. Furthermore, project wizards for Visual Studio are included to help create new analyzer/instrumentation tools and to help create plug-ins to Microsoft's Phoenix C++ compiler backend (C2.exe) [22]. The currently supported languages are C++ and C#.

5.1.1 Is Phoenix the right choice?

Using the current Phoenix RDK release and the tools within it, it should be possible to build an aspect weaver based on the Phoenix framework. This has already been accomplished in the Aspect.NET project [15]. Based on several factors we have however chosen not to use the framework in our project. The primary reason is the steep learning curve to get started using the framework. Currently the documentation for the framework is based upon a dozen of samples mainly in C++ and a help file which is rather inadequate and where most of the code is also in C++. It should be mentioned that we prefer to implement our weaver in C#. The documentation also focus more on how different parts of Phoenix works and not on how to use them. Furthermore, the wizards mentioned previously did not work for C# and the planned tutorials on the Phoenix homepage are not available at the time of writing. Also, the intermediate representation used in the framework is almost too advanced for our needs, giving a lot of unnecessary information that just makes it harder to find the needed data. Last but not least, we have contacted the people behind Aspect.NET asking why many features were missing in Aspect.NET (such as the lack of support for instance method interception and introductions). They replied that this is due to shortcomings in Phoenix. They did not give any further details on the cause of these shortcomings [21].

5.2 Cecil

Cecil [20] is a library written under the Mono Project [16] by Jean-Baptiste Evain. The main functionality of Cecil is to generate and inspect programs and libraries in the CIL format. In practice, this means that it can represent an assembly in a structure with modules, types, methods, instructions, etc. From this structure it can generate a new assembly. It is possible to manipulate, delete and insert objects in the structure. The structure and its objects are in most cases equivalent with the representation of a program in CIL. When Cecil builds a new assembly, it helps the user by taking care of practical low-level issues, like maxstack and offset calculation, name checking on local variables (to avoid name collision on local variables) and checking whether operands to opcodes are of a valid type. More generally it also makes sure that the structure of the assembly is in a correct format.

5.2.1 Is Cecil the right choice?

With the possibilities in Cecil to read, manipulate, create and write new assemblies, it seems to contain the functionality needed to build an aspect weaver. The library is also used in AspectDNG, whose development team included the author of Cecil. As for Phoenix, useful documentation is almost non-existing: There is only a small FAQ which shows a couple of usage examples. However, as the library has a fairly intuitive structure and the focus of the library is relatively narrow (at least compared to Phoenix), we believe it is possible to learn how to use it without any documentation.

Through some small tests we have found that Cecil is relatively easy to learn and that it can be used for at least some of the ideas we have for YIIHAW. The clear correspondence between CIL and the internal structure built by the library also adds an extra plus to Cecil, as it allows making low-level optimizations in the CIL code generated. Based on these observations we have chosen to use Cecil in the development of YIIHAW.

Chapter 6

The complexity of advice inlining

Section 3.4.2 considered how our aspect weaver should work with interceptions. This chapter goes into further details about some of the problems incurred when inlining advice within the target and how we believe they should be handled. All discussions here will be somewhat theoretical; details about the actual implementation are deferred to chapter 9.

6.1 Advice syntax

In section 3.4.2 we defined the need for an advice implementation syntax that allows the user to implement a "catch-all" advice method, i.e. an advice method that can be used for intercepting any kind of target. This was shown by making the advice method return *any type*[1] (represented as *T* in the examples shown in section 3.4.2). This syntax resembles that of AspectDNG where advice methods should always be specified as returning *object*, meaning that they can be used to intercept methods of any type (including *void*).

Looking deeper into what the concrete syntax should look like, one needs to carefully consider what the user can actually write in an advice method. We believe that it should always be possible to overwrite the return value of the intercepted target, as this allows for total replacement of the target. This obviously means that the user should be able to return any type of value from an advice method (as the target methods can be of any type). As mentioned earlier, in AspectDNG this can be achieved by declaring the advice method to return *object* and then return the kind of object you want, as shown in figure 6.1.

```
object Advice(JoinPointContext jpc)
{
  object result = jpc.Proceed(); // invoke the original target
  Console.WriteLine("advice here...");
  return "new return value"; // return a new value
}
```

Figure 6.1: An advice method written using AspectDNG syntax. The *JoinPointContext* object passed as argument to the advice method is an object defined in the AspectDNG framework, that allows the user to obtain information about the intercepted object. In this sample, the object is used to invoke the *Proceed()* method.

As can be seen, this advice method return a string with the value "new return value". This is allowed by the compiler, as a *string* is obviously also an *object*. The problem with this advice

[1] Recall that the examples shown in section 3.4.2 are written as pseudo-code and cannot be directly implemented in any of the .NET languages.

is that it only makes sense to apply it to targets that are defined to return *string*: Inserting this advice into a target that returns *void* would obviously result in a runtime error. The weaver should therefore be able to figure out what type is actually being returned and make sure only to apply the advice to targets of that type (or alternatively, terminate with an error). This kind of typechecking can be very cumbersome to implement, not to say very errorprone. We believe that explictly stating the return type is a better approach, as this makes the typechecking by the weaver unnecessary. Using this approach, the sample method shown in figure 6.1 should thus be explicitly stated as returning *string* instead of *object*.

However, explictly stating the return type removes the possibility for implementing advice methods that can be applied to all targets, regardless of their return type. This was the key intention behind using *object* as a return type in AspectDNG in the first place. As implementing a "catch-all" advice method is often needed (for example, when implementing a logging advice), this feature must obviously be supported by our weaver as well. Thus, a supplemental syntax is needed for these cases. We propose using generics when representing "catch-all" advice methods, as illustrated in figure 6.2.

```
T Advice<T>()
{
  T result = Proceed<T>(); // invoke the original target
  Console.WriteLine("advice here...");
  return result; // return the result from the original target
}
```

Figure 6.2: A "catch-all" advice method written using generics.

As can be seen, this advice can be used to intercept any target, regardless of the return type. The main difference between this syntax and the one of AspectDNG, is that using generics imposes some restrictions on what can be returned from this method: Trying to return a string with the value "new return value" would be illegal in this context and would be discarded by the compiler. The only valid constructs that can be returned from this context are *Proceed()* and *default(T)*[2]. All return types are checked by the compiler, which makes the implementation of the aspect weaver much simpler.

One thing to note about the syntax is that the signature of the advice method is modified slightly, so that the parameter T is explicitly defined on the advice method. This is made in order to avoid complaints from the compiler. The generic type is of course not restricted to be named T - the user can choose whatever name he likes. Similarily, the signature of the *Proceed()* method is modified, so that it is also defined using a type parameter. Again, this is needed in order for the compiler to accept the code. Besides these minor changes, the syntax in figure 6.2 is identical to that found in section 3.4.2.

We believe that the use of generics yields a simple and efficient implementation syntax for the advice methods, as it provides the user with an early typechecking (by the compiler) that catches any attempt to return a type that is incompatible with the target: It is simply not possible to return a specific type on an advice method declared using generics[3]. This makes

[2]This is actually not completely true, as other usages of T can be implemented. However, for the sake of argument assume that no other valid constructs exist at this point. In chapter 9 we will come back to this issue and discuss other uses of the generic type.

[3]This is actually possible if a constraint is set on the generic type. However, defining such constraints means that the advice methods can no longer be used for intercepting any kind of type. Setting a constraint on a generic type is thus the same as explictly defining a specific return type on the advice method.

sense, as you should never be able to overwrite the return value in a "catch-all" advice method, as it is simply impossible to return a type compatible with all targets. If the user needs to overwrite the return type this can still be done by explicitly defining a return type on the advice method. This advice method is then used only for intercepting targets with that return type.

6.2 Invoking Proceed

Invoking the *Proceed()* method should trigger the weaver to inject the original target at the specified location. This is somewhat complicated by the fact that the weaver need to consider the context in which the original target is injected. Consider the advice shown in figure 6.3.

```
T Advice<T>()
{
  T result = Proceed<T>();
  Console.WriteLine(''advice here...'');
  return result;
}
```

```
.locals init ([0] !!T result)
ldarg.0
call        instance !!0 YIIHAW::Proceed<!!0>()
stloc.0
ldstr       "entering advice..."
call        void [mscorlib]System.Console::WriteLine(string)
ldloc.0
ret
```

Figure 6.3: An advice method represented as C# code (top) and IL Assembler language (bottom).

The advice method invokes *Proceed()* and stores the result in a variable named *result* of type *T*. As the advice method is declared to return a generic parameter, it can be used for intercepting any kind of target. However, when applying this advice to the intercepted method it no longer makes sense to store the result in a generic parameter, as this generic parameter only exists in the advice method. In most cases it would actually result in a runtime error if the generic parameter is introduced within the target method, as the generic parameter will most likely not be defined within the declaring type of the target method. The generic parameter should thus be replaced with the actual type of the target method being intercepted. Intercepting a method that returns an *int* should thus trigger the weaver to modify the local variable *result* from type *T* to type *int*. All other local variables that store the result of *Proceed()* should be modified as well. These modifications should of course be made for all types.

6.2.1 Handling void

Special care needs to be taken when handling interception of target methods that returns *void*. It makes no sense to modify the generic parameter to be of type *void*, as this is not a valid type. Thus, a different approach is needed when handling these methods. As a target method of type *void* obviously never returns anything, it makes sense to skip the assignment to the generic parameter altogether. This also implies that loading the value of the generic parameter at a later stage should also be skipped, as it will not have been assigned a value in the first place. Figure 6.4 shows a modified version of the advice from figure 6.3, transformed by the aspect weaver in order to handle *void* methods.

```
ldarg.0
call        instance !!0 Weaver::Proceed<!!0>()
ldstr       "entering advice..."
call        void [mscorlib]System.Console::WriteLine(string)
ret
```

Figure 6.4: An advice method modified by the aspect weaver in order to handle interception of target methods that returns *void*.

As can be seen, the *stloc* and *ldloc* instructions have been skipped. There is no need for them when intercepting a *void* method, as there is no value to store and load. Similarily, as the variable *result* is no longer needed, this can be skipped as well. There is no need for introducing a variable into the intercepted target method that is never used. The advice method shown in figure 6.4 is still valid, i.e. the CLR will have no problem executing it, even though some of the instructions have been removed. These instruction were only used for storing and loading the result of *Proceed()* anyway. However, as we know that the intercepted target method returns *void*, we also know that the call to *Proceed()* will place no value on the stack. Thus, when reaching the return statement (*ret*) no value is left on the stack, which is just what is intended when intercepting a *void* method.

6.2.2 Injecting the target method's body

So far, we have only discussed how the advice method should be modified in order to fit into the context of the methods being intercepted. When handling a call to *Proceed()* the weaver should do more than just changing the type of the variables that store the result returned from *Proceed()*. Invoking *Proceed()* is actually a cue to the weaver that it should at this point inject the original target method's body into the code being generated, substituting the call to *Proceed()*. Consider the sample target method in figure 6.5. Intercepting this method using the advice shown in figure 6.3 should result in a new target method, consisting of the advice method merged together with the original target method, as shown in figure 6.6.

```
int Target()
{
  System.Random random = new System.Random();
  return random.Next();
}
```

```
.locals init ([0] class [mscorlib]System.Random random)
newobj      instance void [mscorlib]System.Random::.ctor()
stloc.0
ldloc.0
callvirt    instance int32 [mscorlib]System.Random::Next()
ret
```

Figure 6.5: A target method represented as C# code (top) and IL Assembler language (bottom).

The generated code is fairly simple: As the intercepted method returns an *int*, the local variable *result* is set to be of this type as well and the variable is introduced into the intercepted target method (as can be seen in the *.locals* directive). The call to *Proceed()* is replaced with all instructions from the original target method. Following the last instruction of the original target method, is a *stloc.0* instruction that takes the *int* value returned from the original target

```
.locals init ([0] int32 result, [1] class [mscorlib]System.Random random)
newobj     instance void [mscorlib]System.Random::.ctor()
stloc.1
ldloc.1
callvirt   instance int32 [mscorlib]System.Random::Next()
ret
stloc.0
ldstr      ''advice here...''
call       void [mscorlib]System.Console::WriteLine(string)
ldloc.0
ret
```

Figure 6.6: The result of weaving the target method of figure 6.5 using the advice of figure 6.3.

method and stores it in variable 0 (named *result*). This variable is then loaded again later on and is returned.

Handling return statements

One major problem exist in the code shown in figure 6.6: Two return (*ret*) instructions exist in the generated output. This is a valid .NET program, but it will lead to undesired behaviour, as the method would return as soon as it reached the first of these two *ret* instructions. This means that all instructions following the original target method will never be executed. This problem is due to the *ret* instruction located in the original target method, which is also inserted into the generated output along with the other instructions. As the new target method should never return before all the advice instructions have been executed, it is necessary to remove these *ret* instructions and defer returning until reaching the end of the new target method. However, this has to be done with care, as simply moving the *ret* instructions significantly changes the behaviour of the original target method. Figure 6.7 shows an example of this: Removing the *ret* instruction would cause the execution to fall-through to the *ldstr* instruction and print "b is not true" no matter what the value of *b* is. This is clearly not what was intended, so another approach is needed.

```
void SomeMethod(bool b)
{
  if(b)
    return;

  Console.WriteLine(''b is not true'');
}
```

```
ldarg.1
brfalse.s      label
ret
label: ldstr   "b is not true"
call           void [mscorlib]System.Console::WriteLine(string)
ret
```

Figure 6.7: A method represented as C# code (top) and IL Assembler language (bottom).

We propose replacing all *ret* instructions within the original target method with a *br* instruction, that unconditionally branches to the instruction following the last instruction of the original

target method. This effectively prevents any premature returns that changes the behaviour of the original target.

6.3 Merging local variables

Figure 6.6 showed the generated output when applying the advice of figure 6.3 to the target method defined in figure 6.5. Merging the local variables is actually a bit more complex than illustrated in this figure, as it is necessary to update all references to these variables as well. Notice how the references (*stloc* and *ldloc* instructions) from the target method have been updated to refer to the new index of the *random* variable in figure 6.6. This needs to be done for all variables whenever local variables are merged together from the advice and from the original target. The weaver should use the short version of these instructions whenever possible in order to optimize the generated code.

An important issue when merging the local variables is that the variables from the original target method should only be included in the generated code if *Proceed()* is invoked at some point within the advice method. If the *Proceed()* method is never invoked this means that the user intend to completely replace the intercepted target method. In this case, it makes no sense to include the local variables of the original target, as they will never be used.

6.4 Mapping IL-instructions

During the previous sections we have discussed various modifications that need to be performed on the target methods being intercepted and on the advice methods applied to them. These modifications affect not only the instructions that are directly manipulated - they also affect other instructions that refer to these modified instructions. Consider the example shown in figure 6.8: A boolean variable, *b*, is introduced which gets assigned a value (the actual value is irrelevant for this discussion). If *b* is *true* the *Proceed()* method is invoked. In IL Assembler language, such an expression is implemented using a branching instruction (*brfalse.s* in this sample) that branches to a particular instruction if the proper condition is met. In figure 6.8 the instruction that is targeted by the branching instruction is illustrated using a label. However, the actual IL code generated does not contain any labels. Instead this instruction is calculated as an offset to the branching instruction. This poses a problem when replacing the call to *Proceed()* with the actual instructions of the target method, as this naturally changes the offset of the instruction that should be branched to as well.

In order to solve this problem a mapping needs to be maintained: When modifying an instruction, a reference to the old instruction and the modified instruction should be stored in memory. Having modified the instructions, all branching instructions should be updated according to the mapping. This way, no "dangling" references exist that might result in runtime exceptions when invoking the generated assembly.

6.5 Checking references

In the examples shown so far we have not considered how to handle references to different kinds of constructs (methods, fields, classes, etc.) from within the advice methods. These references have to be checked by the weaver to make sure that they are valid within their new context. Consider the sample code shown in figure 6.9: The call to *SomeMethod()* should be checked by the weaver in order to make sure that this method is available from the target assembly. If this

```
public T Advice<T>()
{
  bool b = GetBool();

  if (b)
    Proceed<T>();

  Console.WriteLine("advice here...");
  return default(T);
}
```

```
.locals init ([0] bool b,
          [1] !!T CS$0$0000)
ldarg.0
call          instance bool Aspects::GetBool()
stloc.0
ldloc.0
brfalse.s     label
call          !!0 YIIHAW::Proceed<!!0>()
pop
label: ldstr  "advice here..."
call          void [mscorlib]System.Console::WriteLine(string)
ldloca.s      CS$0$0000
initobj       !!T
ldloc.1
ret
```

Figure 6.8: An advice method represented as C# code (top) and IL Assembler language (bottom). Replacing the call to *Proceed()* with the instructions of the target methods requires that the *brfalse.s* instruction is modified so that it points to the correct offset.

check is not performed, this call might result in a runtime exception being thrown by the CLR when executing the program. This type of checking is somewhat complicated by the fact that the action that should be taken by the weaver depends on the context in which the construct is used: Three different types of contexts can be defined. These will be discussed in the following sections.

```
class Aspects
{
  T Advice<T>()
  {
    Foo.SomeMethod();
  }
}
```

Figure 6.9: An advice method that invokes a static method, *SomeMethod()* on class *Foo*.

6.5.1 Handling references to constructs outside the aspect assembly

Consider the sample shown in figure 6.9. If the class *Foo* is located outside the aspect assembly (i.e. outside the assembly in which the advice method is defined) the aspect weaver should check that this class is available from the target assembly. This implies that the weaver should check that the assembly in which *Foo* is defined is available from the target assembly. This can easily be checked, as the manifest of the target assembly should contain a reference to this assembly.

It this is not the case, the weaver should add a reference to the assembly, as the call would otherwise result in a runtime exception being thrown when invoking the program. A similar action should be performed when handling references to fields, properties, classes, interfaces and delegates.

6.5.2 Handling references to constructs in the aspect assembly

Special consideration should be taken when handling references to constructs within the aspect assembly itself. As all constructs within the aspect assembly might be subject for insertion into the target assembly (via the pointcut), this means that ambiguous references might be encountered. Assume that the class *Foo* of figure 6.9 is defined within the aspect assembly instead and the call to *SomeMethod()* from the advice method refers to this class. The user might instruct the aspect weaver to insert *SomeMethod()* into all classes in the target assembly. This is a perfectly legal pointcut. However, this might result in a scenario where *SomeMethod()* suddenly exists in 30 different classes in the target assembly. There is no way for the aspect weaver to determine which of these 30 classes to use when invoking *SomeMethod()* - the reference is ambiguous in the new context. It does not make any sense to just add a reference to the aspect assembly and refer to *SomeMethod()* defined in this assembly, as this would mean that the target assembly is now dependent on the aspect assembly when invoking the program. This type of dependency should be avoided, as the aspect assembly is created as an ad hoc assembly that should only be used by the aspect weaver - it makes no sense on its own.

We propose the following solution: The aspect weaver should check whether *SomeMethod()* is instructed to be inserted into the target assembly. If this is not the case, the weaver should terminate with an error, as this means that *SomeMethod()* is not available at all from the target assembly. If *SomeMethod()* on the other hand is subject for insertion, the weaver should check the number of locations in which the method is inserted. If the method is only inserted into one class, this means that the reference is not ambiguous and the weaver should replace all calls to *SomeMethod()* with its new location within the target assembly. If the method is inserted at multiple locations, the weaver should terminate with an error, as this obviously means that the reference is ambiguous. Similar action should be performed for other constructs refered to from the advice methods.

6.5.3 Handling references to local constructs

Consider the sample shown in figure 6.10. This sample is similar to that of figure 6.9, except that the call to *SomeMethod()* is now a local call, i.e. *SomeMethod()* is defined within the same class as the advice method.

Methods (and properties) often refer to other local constructs defined within the same object as the method itself. In figure 6.10 the method *SomeMethod()* updates the value of the local field *x*. Such references to local constructs should be maintained when applying the aspects. This implies that the invocation of *SomeMethod()* shown in figure 6.10 should be mapped to the local version of this method within the target class. Figure 6.11 shows the result of applying the advice to a target method.

As can be seen, the local field *x* and the local method *SomeMethod()* have been inserted into the target class[4]. The method that was intercepted (*TargetMethod()*) invokes the local version

[4]Here we assume that these constructs were targeted by the pointcut defined by the user. If the user does not explicitly state that these constructs should be introduced within the target assembly, the weaver should terminate with an error, as the references to these constructs would be invalid.

```
class Aspects
{
  int x;

  void SomeMethod()
  {
    x = 7;
  }

  T Advice<T>()
  {
    SomeMethod();
  }
}
```

Figure 6.10: An advice method that invokes a local method, *SomeMethod()* which updates the field *x*.

```
class Target
{
  int x; // field inserted by the aspect weaver

  void SomeMethod() // method inserted by the aspect weaver
  {
    x = 7;
  }

  void TargetMethod() // method intercepted by the aspect weaver
  {
    SomeMethod(); // call the local version of SomeMethod()
  }
}
```

Figure 6.11: The result of applying the advice method of figure 6.10 to a method in the target assembly. All local references within the aspect assembly are maintained in the target assembly. (Note that this example is only shown in C#, as it is easier to understand. The weaver never produces source code, as it operates at the assembly-level.)

of *SomeMethod()*. This is an important thing to notice. The method *SomeMethod()* might be subject for insertion into many locations in the target assembly. However, local references within the advice methods should always be mapped to local references within the target assembly as well. This means that *SomeMethod()* might be inserted at 30 different locations within the target assembly. This is no problem (that is, the reference is not ambiguous) as long as all references to this method are local references. Local references thus form an exception to what was defined in the previous section, which stated that if a construct is inserted at multiple locations, the references to it are ambiguous. The weaver should be able to make these transformations of local references.

6.6 Referring to the declaring type of the target method

When implementing an advice method, one often need to access the declaring type of the method being intercepted, e.g. to access fields or invoke methods on this type. Such a scenario is depicted in figure 6.12.

```
class TargetClass
{
  int x = 5; // this field should be accessible from the advice method

  int TargetMethod() // this method should be intercepted and the value of 'x'
      should be returned instead
  {
    return 3;
  }
}
```

Figure 6.12: A class containing one method, *TargetMethod()*, that should be intercepted. The advice that should be applied need to access the field *x*.

As can be seen, the declaring type of the method that should be intercepted, *TargetClass*, contains an instance field, *x*. The advice method should be able to access this field and use it (either to read the value or update the value) within the advice. In many aspect weavers, including the ones examined in chapter 2, this can be achieved by accessing a special property (most often named *Target*, *DeclaringType* or something similar). As the declaring type of the method being intercepted can obviously be of any type, these properties are defined to return *object* in order to be able to support all declaring types. As one often need to access methods or fields specific to the type being intercepted a typecast need to be performed. This is illustrated in figure 6.13.

```
object Advice(JoinPointContext jpc)
{
  TargetClass target = (TargetClass)jpc.RealTarget; // get a reference to the
      declaring type of the target method and typecast it

  return target.x; // return the value of the field 'x'
}
```

Figure 6.13: An advice method written using AspectDNG syntax that access the declaring type of the method being intercepted and returns the value of the instance field *x*.

We consider this kind of syntax to be a bad approach, as the use of typecasts adds runtime overhead. In our case, the only purpose of the typecast would be to please the compiler: As the advice gets merged with the original target method at weave-time, all constructs referred to from the advice methods are directly available in the new context. This means that the typecast is completely unnecessary once the weaver has applied the advice. For this reason, we propose a different syntax: Use generics to specify the type of the target class. This is shown in figure 6.14.

```
T Advice()
{
  TargetClass target = GetTarget<TargetClass>(); // get a reference to the
      declaring type of the target method

  return target.x; // return the value of the field 'x'
}
```

Figure 6.14: An advice method that performs the same operation as shown in figure 6.13 by using a syntax based on generics.

The *GetTarget()* method is defined to take a type parameter of the same type as the the target class. This syntax resembles that of *Proceed()* described earlier. The *GetTarget()* method returns the instance of the target class that is being intercepted. No typecasts are needed for accessing fields and methods of the declaring object, as the type is explicitly defined when invoking the method.

6.6.1 Handling calls to *GetTarget()*

At weavetime, all calls to *GetTarget()* should simply be replaced with a *ldarg.0* instruction, i.e. a *this* pointer. This is possible, as our weaver uses an inlining approach that effectively transfers all instructions of the advice methods into the target methods. In this new context a reference to the declaring object can be obtained directly via a *this* pointer. The use of *GetTarget()* should obviously only be allowed when intercepting instance methods. This should be checked by the weaver. Similarily, the weaver should check that the type specified when invoking *GetTarget()* is equal to the actual declaring type of the method being intercepted. If this is not the case the weaver should terminate with an error.

6.6.2 Accessing constructs of the declaring type

As the target and the advice methods are located in separate assemblies, one problem exists with the syntax proposed in figure 6.14: All constructs that should be available from the advice methods need to be accessible outside the assembly in which they are defined. Constructs defined as being *public* obviously poses no problem, as these can be accessed from any class in any assembly. However, constructs defined as *internal*, *protected* or *private* are somewhat more difficult to handle, as these cannot be directly accessed from another assembly. No perfect solution exist for this problem, as it is bounded by a fundamental design decision in the .NET framework. An obvious solution would be to change the access specifier to *public* for constructs that need to accessed from the advice methods. However, such an approach is not always suitable, as this would break the principle of encapsulation. For these cases, we propose changing the access specifier of the constructs to *internal* or *internal protected* instead and use the *InternalsVisibleToAttribute* attribute[5], specifying that the aspect assembly should be able to access internals in the assembly. This way, all access to the constructs are prohibited outside the declaring assembly, except by the aspect assembly. This is the same approach as used in the "Generation of specialized collection libraries" project [3]. This problem exist in AspectDNG, Aspect.NET and Rapier LOOM as well. NKalore does not have this problem, as it uses preprocessing weaving, i.e. the advice is applied before compiling which eliminates the problem.

[5]http://msdn2.microsoft.com/en-gb/library/system.runtime.compilerservices.internalsvisibletoattribute.aspx

Chapter 7

Pointcut specification

So far, all discussions have focused on various facets of implementing a high-performing aspect weaver. However, these discussions have all been somewhat theoretical in the sense that they did not describe any concrete implementation details. Starting with this chapter, focus is now directed towards the actual implementation of our aspect weaver. This chapter is going to present details related to the implementation of the pointcut parser. All discussions will be based on the ideas described throughout the preceeding chapters. A class diagram showing the main entities and relations can be seen in appendix D.

7.1 Defining the pointcut language

One of the most important things to consider when implementing a pointcut parser is defining what the pointcut language should look like. The pointcut language serves as the user's primary way of interacting with the weaver, as it is what basically allow the user to specify which aspects should be applied to the target assembly.

7.1.1 The pointcut file

Recall from section 3.5 that we consider using a separate file for the pointcuts to be the best approach, as this allows greater flexibility when it comes to defining the pointcuts. An issue in this regard is deciding on the format of the pointcut file. An obvious idea is to use XML for writing the pointcuts, as it is well-structered and fairly easy to read and write. AspectDNG uses XML for defining the pointcuts. However, we believe that XML adds too much "noise" when writing the pointcuts: Looking at a typical input file for AspectDNG, more than half of the input consist of the definition of tags and properties that are necessary[1] to include in order to specify the real concern, namely the pointcut itself. We consider using an ad hoc aspect language to be a more suitable approach, as this allows for a syntax where the definition of these unnecessary constructs can be avoided. Aspect.NET also uses such an approach.

7.1.2 Targeting constructs

Section 3.5.2 looked at some of the expressions that should be supported by our weaver. Three types of expressions should be supported: Interceptions, introductions and typestructure modifications.

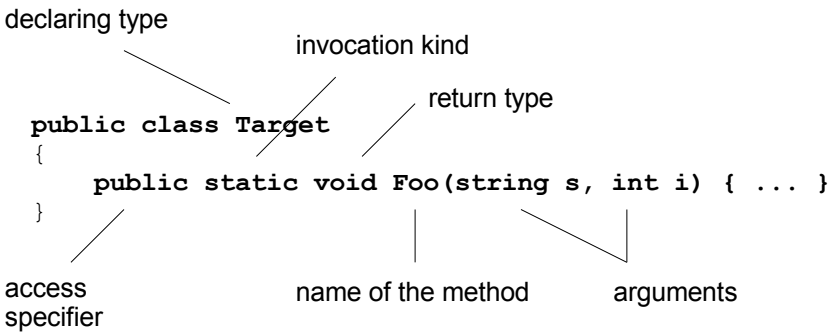
[1] The sample pointcut files in appendix G gives an idea of what the AspectDNG pointcut files look like.

Interceptions

As mentioned in chapter 3, our weaver should only support *around body* interceptions. To give the user a fine-grained control of which constructs should be targeted the pointcut language should support the specification of the following properties on the target constructs:

- Access specifier
- Invocation kind (instance/static)
- Return type
- Name of the method
- Arguments
- Declaring type of the construct (namespace and class)

These properties allow the user to match the following method by a pointcut statement:



As the user should be able to write a single pointcut that targets many constructs, the use of wildcards should be supported on all these properties.

Introductions

The weaver should support introducing various types of constructs into the target assembly. The following constructs should be supported by the pointcut language:

- Methods
- Properties
- Fields
- Classes
- Delegates
- Events
- Attributes
- Enumerations

Classes, attributes and enumerations can be inserted into an existing namespace or class (which means that they are nested inside the specified class) in the target assembly. All the other constructs can only be inserted into an existing class. The pointcut language should support the specification of the following properties on the constructs:

- Construct type (one of six types listed above)
- Access specifier
- Invocation kind (instance/static)
- Return type (methods, properties, fields, events and delegates only)
- Declaring type of the construct (namespace and possibly class)
- Declaring type of the target class or namespace (the location where the construct should be introduced)

As for interceptions, the use of wildcards should be supported on these properties.

Typestructure modifications

The weaver should support two types of typestructure modifications: Interface implementation and changing the superclass of an existing class. The following properties should be supported by the pointcut language:

- Declaring type of the target class (the class that should be modified)
- Type of modification (implementation of an interface or changing the superclass)
- Declaring type of the new interface or superclass

Once again, the use of wildcards should be supported.

7.1.3 The pointcut language

The grammar for the pointcut language that we propose can be seen in appendix B. Figure 7.1 shows the syntax that can be derived from this grammar. Figure 7.2 shows examples of the pointcut syntax. As the pointcut syntax is easier to grasp than a complete grammar, we will use this as a foundation when describing the pointcut language.

around

All interception-statements start with the keyword *around*. Properties declared before the keyword *do* describes the target methods to intercept. Properties succeeding the keyword *do* describe the advice method to use. Wildcards are only allowed when specifying the target methods - specification of the advice method should always match the name of a specific method. A single method can be targeted multiple times (using multiple statements). In that case, the method is intercepted in the same order as the statements are declared.

The *access* property allows specifying the access specifier for the method to intercept. Possible values are: *public*, *private*, *protected*, *internal* or ***.

The *invocation kind* property can be used for specifying the invocation kind of the method to intercept. Possible values are: *static*, *instance* or ***.

Types and arguments must be fully qualified (namespace and class). An exception to this rule is the primitive datatypes found in the .NET framework, such as integers, string, chars, etc. These can be written using a short form notation similar to the syntax of C#. Details about these notations can be found in appendix C. If more than one argument is specified on the target method, arguments must be separated with a comma. Alternatively, one can use a wildcard (*) to match any number and any types of arguments.

The *inherits* keyword is an optional property that allows specifying restrictions on the superclass of the declaring type of the target method. Figure 7.2(b) shows a sample usage of the *interhits* keyword that matches all methods whose declaring type inherits from *System.Collections.Hashtable*. Leaving out the specification of the *inherits* property is the same as writing *inherits **.

Notice that the advice method is specified in a different manner than the target methods. As can be seen in figure 7.1, the access specifier, invocation kind, return type and arguments are not specified for advice methods. The reason for this is as follows: The weaver allows the user to write multiple advice methods with the same name (of course, the signature must be unique, as the code would otherwise not be compilable). When probing the target and aspect assemblies for methods that matches the pointcut, the weaver automatically picks the advice method whose signature provides the best match for each target method. This means that the user can write a single pointcut that matches many (even all) methods in the target assembly and apply different advice for each target (depending on the number and the type of advice methods he has implemented). We will go into further details about this subject in the next chapter.

insert

Introductions start with the keyword *insert*. Properties preceeding the keyword *into* describe the construct that should be inserted. This construct must be located in the aspect assembly. The property succeeding the keyword *into* specify where the construct should be inserted.

The property *construct* is used to specify the type of construct that should be inserted. Possible values are: *method*, *property*, *field*, *event*, *delegate*, *enum*, *attribute* and *class*. No wildcards are allowed on this property.

The properties *access*, *invocation kind* and *return type* are similar to the ones defined for interceptions. The *return type* property should not be specified when introducing a class, enum or attribute (as these do not have any return type). For all other constructs the return type must be specified. Wildcards are not allowed on this property.

The *aspect type* and *aspect name* properties must be fully qualified. No wildcards are allowed on these properties, as the user should always specify a concrete construct that should be inserted.

The last property (*type*) defines the type in which the construct should be inserted. Wildcards are allowed on this property, which means that the user can instruct the weaver to insert a construct at various locations in the target assembly using one single statement (see figure 7.2(e) for an example of this).

modify

Typestructure modifications are defined using the *modify* keyword. The *type* property specifies the type to modify. This type must be fully qualified and must be located in the target assembly.

The *action* property defines the type of action to perform. The possible values are: *inherit* and *implement*. If the former is chosen, the *aspect type* property must define a fully qualified class within the aspect assembly. If the *action* property is set to *implement*, the *aspect type* property must define a fully qualified interface in the aspect assembly. The weaver will check if all methods and properties of that interface are defined within the target type. If this is not the case, the weaver will terminate.

Interceptions:

```
around <access> <invocation kind> <return type> <type>:<method(arguments)>
[inherits <type>] do <advice type>:<advice method>;
```

Introductions:

```
insert <construct> <access> <invocation kind> [return type]
<aspect type>:<aspect name[(arguments)]> into <type>;
```

Typestructure modifications:

```
modify <type> <action> <aspect type>;
```

Figure 7.1: Syntax for the pointcut language. Constructs in square brackets are optional. Constructs in angle brackets are mandatory.

7.2 Scanning the input file

Having defined the pointcut language, we now turn focus towards the actual implementation of the pointcut parser. Recall from previous discussions that all pointcuts should be defined in a separate textfile. This obviously requires that the file at some point gets loaded into memory and that the content is divided into smaller fragments that can be parsed. Usually such a task is handled by a *tokenizer*. Unfortunately the .NET framework does not include a tokenizer, so we had to build one of our own (see appendix V for the source code). The tokenizer (fully qualified name in the source code: *YIIHAW.Pointcut.LexicalAnalysis.Tokenizer*) is very simple: It splits the text into tokens and determines its type. The type can be either *EOF (END-OF-FILE)*, *WORD* (string) or *NUMBER*. If the token is of type *WORD* the string value can be fetched using the property *Sval*. If the token is a *NUMBER* the integer value can be fetched using the *Nval* property.

7.2.1 Identifying keywords and special tokens

Having only three types of tokens when implementing a parser means that various checks have to be performed in the parser in order to identify keywords and other constructs. This type of checking is repetitive and error-prone. A suitable solution for this problem is to identify all special tokens (keywords and strings) prior to parsing the content. This process is refered to as

Interceptions:

```
(a) around public static void TargetNamespace.TargetClass:Foo(int,string)
    do AdviceNamespace.AdviceClass:AdviceMethod;

(b) around * * *:*(*) inherits System.Collections.Hashtable
    do AdviceNamespace.AdviceClass:AdviceMethod;
```

Introductions:

```
(c) insert method public * int Namespace.AspectClass:Foo(string,System.Object)
    into TargetNamespace.Class;

(d) insert class Namespace.AspectClass into TargetNamespace;

(e) insert delegate Namespace.AspectClass:MyDelegate(int,string)
    into TargetNamespace.*;
```

Typestructure modifications:

```
(f) modify TargetNamespace.Class inherit Namespace.AspectClass;

(g) modify TargetNamespace.Class implement Namespace.AspectInterface;
```

Figure 7.2: Examples of the pointcut syntax.

scanning [14].

We have implemented a scanner (fully qualified name: *YIIHAW.Pointcut.LexicalAnalysis.Scanner*) that takes care of identifying all special tokens. Two types of tokens exist: Keywords (such as *around*, *insert*, *do*, etc.) and special characters (such as "(", ")", ":", ";", "*", etc.). In total, 31 types of tokens exist in the pointcut language (refer to the source code in appendix V for details about these tokens). All of these tokens are defined as values of an *enum* value (called *TokenType*), which means that the parser can determine the type of token by looking at the value of this *enum* - the parser does not need to identify the type of token itself.

A special type of token called *NAME* exist: This token contain a string literal used by the user to identify a construct in the target or aspect assembly (for example, the return type of a method). For these tokens, acquiring the token type is not enough; the parser must be able to fetch the actual string literal of the token as well. This can be done using the *Sval* property of the scanner.

7.3 Parsing the pointcuts

Having defined the pointcut grammar and identified all tokens, the implementation of the parser is fairly simple: A *parsing method* [14] is defined for each *nonterminal symbol*[2] in the pointcut

[2]A nonterminal symbol is a special kind of symbol that can be resolved (often recursively) as one or more terminals (strings). Nonterminals are only used for making it easier to define and implement a grammar. They can never appear in the text entered by the user - only terminals can.

grammar. As the pointcut grammar contain 22 nonterminals this means that 22 parsing methods are defined within the parser (fully qualified name in the source code: *YIIHAW.Pointcut.Parser*). For details regarding these methods, refer to the source code in appendix V.

7.3.1 Storing the pointcut statements

As the parser is not responsible for performing the actual weaving of the aspects, this means that the pointcut statements must somehow be stored for later retrieval by the weaver. Three structures which represents the abstract syntax for the pointcut language, are defined for this purpose: *Around*, *Insert* and *Modify*. These structures contain fields and properties for storing and retrieving the details of a single pointcut statement. For instance, the *Around* structure contain properties, such as *Access*, *ReturnType*, *TargetType*, etc. for storing details related to an interception statement. Every single statement gets stored into one of these structures (which structure obviously depends on the type of pointcut statement). Once the parsing has been performed the pointcuts can be retrieved using the properties *AroundStatements*, *InsertStatements* and *ModifyStatements* on the parser object. Each of these properties return a strongly typed collection.

7.3.2 Handling errors

As the user does not always specify valid pointcut statements the parser must obviously be able to identify and handle invalid input. Each parsing method checks that the current token is as expected. If the parser at any point identifies a token that was not expected an exception is thrown. A sample exception might be: `Expected ';', but found 'insert'`. All exceptions thrown by the parser are catched and handled by the weaver.

Chapter 8

Identifying targets and aspects

Having parsed all the pointcut statements the process of identifying the constructs that match these statements can begin. This is handled by three different classes: *InsertHandler*, *ModifyHandler* and *InterceptHandler* (all located in the namespace *YIIHAW.Controller*). The source code for these classes can be seen in appendix W.

8.1 Locating constructs in the target and aspect assemblies

Locating constructs in the target and aspect assemblies is fairly simple, as Cecil directly exposes all of the necessary properties that describe these constructs. For instance, the *MethodDefinition* class from the Cecil API provides access to various information related to a single method, such as the name of the method, its return type, type of arguments, etc. Similar classes exists for other types of constructs (fields, classes and so on). By looping over all constructs in the target and aspects assemblies, constructs that match the pointcut statements can easily be located by comparing these properties to the pointcut statements. If the aspect construct cannot be located, an exception is thrown.

8.2 Locating the proper advice method

As mentioned in the previous chapter, no signature is specified for the advice methods when writing interception statements - only the declaring type and name of the advice method are specified. This allows the user to match multiple target methods using a single statement, while still being able to achieve fine-grained control of the interception process (by defining alternative advice methods that specifically match the signature of some of the target methods). An advice method can match a target method in two ways: By the return type and/or by the method arguments. Matching the return type always takes precedence over the method arguments. Consider the following advice methods:

```
public static int Advice() { ... }

public static T Advice<T>(string s, double d) { ... }

public static T Advice<T>() { ... }
```

The first advice method would match all target methods that return an integer regardless of their arguments (as this advice method directly match the return type of such target methods, and an advice with zero arguments always matches any number and any type of arguments of the target methods). The second advice method matches target methods that take a string and

a double as the first two arguments (possibly followed by one or more arguments). This advice method would thus match the following target methods:

```
public double TargetA(string s, double d) { ... }

public string TargetB(string s, double d, int i, float f) { ... }
```

The third advice method would match all target methods not matched by any of the first two advice methods.

The *InterceptHandler* class identifies all target methods that match the signature specified in the pointcut statements. For each match, it determines which advice method should be applied to it. A warning is shown if no suitable advice method can be found.

8.2.1 Invocation kind of the advice methods

The invocation kind of the advice method also has an impact on the process of locating the correct advice method for a given target method. Advice written as an instance method can only be used for intercepting targets that are instance methods, as one might otherwise encounter references that make no sense in the new context. Consider the following example:

```
public class Aspects
{
  public void Foo { ... }

  public T Advice<T>()
  {
    ...
    Foo();
    ...
  }
}
```

This advice method invokes method *Foo* (which is an instance method). The method *Foo* should obviously be inserted into the target assembly so it can be invoked from the intercepted methods. However, if the method being intercepted is a static method, invoking *Foo* would produce illegal code, as you cannot invoke a local instance method from a static context. For this reason, instance advice methods should only be used for intercepting instance methods. Static advice methods does not have this problem, as you can never invoke a local instance method from a static advice method. This means that static advice methods can be used for intercepting both instance and static methods.

When determining which advice method to apply to a given target method, the *InterceptHandler* class make sure that invocation kind of the advice method are compatible with the target method.

Chapter 9

The weaving process

The previous chapter described how targets and aspects are identified. This chapter goes into details about how the actual weaving takes place. All discussions are based on the preliminary ideas described throughout this report. All source code related to the implementation of the weaver can be found in appendix X.

9.1 Introducing constructs

Introducing a construct within the target assembly obviously requires the weaver to insert the construct at the specified location. This process is not as simple as it may sound, as the weaver has to take care of references to other constructs inside the inserted construct. For instance, a method might refer to some method declared within the same class as the method itself. Consider the example shown in figure 9.1.

```
class Aspects
{
  public void Foo()
  {
    Bar(); // invoke method Bar(), which is located in the same class as Foo()
  }

  public void Bar()
  {
    ...
  }
}
```

```
ldarg.0
call       instance void AspectNamespace.Aspects::Bar()
ret
```

Figure 9.1: A method, *Foo()*, represented as C# code (top) and IL Assembler language (bottom).

The method *Foo()* invokes method *Bar()*, which is located within the declaring type of *Foo()*. Looking at the IL Assembler code generated, it can be seen that this method call is fully specified: The CLR is instructed to invoke method *AspectNamespace.Aspects::Bar()*. However, this call makes no sense in the new context, as *AspectNamespace.Aspects* is not available in the target assembly. For this reason, YIIHAW needs to translate all references into the proper context. The method call shown in figure 9.1 should thus be modified to *<TargetNamespace.TargetClass>::Bar()* prior to inserting it into the target assembly. This also requires that

a check is made in order to determine whether the method *Bar()* is available from the target assembly, i.e. if *Bar()* is itself subject for insertion. If this is not the case, the weaving should be aborted.

9.1.1 The need for a two-pass approach

Simply translating references into the proper context is not enough in all cases, as some methods or properties might contain mutually dependent references. Consider the example shown in figure 9.2.

```
class Aspects
{
  public void Foo(int x)
  {
    Bar(x);
  }

  public void Bar(int x)
  {
    if(x > 5)
      Foo(--x);
  }
}
```

Figure 9.2: Two methods that are mutually dependent.

Method *Foo()* and *Bar()* are mutually dependent, as they refer to each other. This is a problem, as Cecil requires that all constructs need to be defined before they can be referenced. This means that method *Foo()* must be defined, before it can be referenced by *Bar()*. Similarily, *Bar()* must be defined before it can be referenced by *Foo()*. This requirement would cause the weaver to deadlock, as there is no way to implement both methods at the same time.

In order to circumvent this problem, we use a two-pass approach when introducing new constructs into the target assembly. On the first pass the construct is defined within the target assembly, but no references to other constructs are inserted into the construct. For methods and properties this means that the construct is created in the target assembly, but none of the IL-instructions are added at this point and their return types are not defined or updated, thereby avoiding references to other constructs. In the second pass the IL-instructions are inserted into the methods and properties created in the first pass. Furthermore, all references are translated to their new context. This is possible, as all constructs exist at this point, which means that deadlocks are avoided.

9.1.2 Storing constructs in a mapping-table

At the first pass, all constructs that are created within the target assemly are added to two mapping-tables. Storing all constructs in mapping-tables allows for easy retrieval of these constructs at a later point. This is useful, as introduced constructs are often referenced from advice methods (or from other methods or properties that are subject for insertion). This means that the weaver must be able to retrieve these constructs again at some point.

Local mapping

Recall from section 6.5.3 that references to local constructs (constructs defined within the same class as the referrer) should be maintained. This implies that one construct defined within the aspect assembly might be inserted at 30 different locations in the target assembly. Each local reference to such a construct should be substituted with a reference to the local version of this construct.

A special class, called *LocalMapper<T>* (fully qualified name in the source code: *YIIHAW.Weaver.LocalMapper<T>*) is defined for handling this. The generic parameter T, specifies the kind of construct to map (we only use the types defined within the Cecil API, such as *MethodDefinition*, *FieldDefinition*, etc.). Given a reference to the target type and a reference to the construct defined in the aspect assembly, the *Lookup()* method returns a reference to the corresponding construct in the target assembly:

$$(TargetType, AspectConstruct) \longrightarrow TargetConstruct$$

A combined key is thus used when invoking the *Lookup()* method. This is necessary as the substituted refererence depends on both the type in which the reference is defined and the construct that is referenced. Using this mapper it is possible to substitute all references to local constructs with the appropriate construct in the target assembly.

Global mapping

Constructs that refer to constructs defined outside their own declaring type, but inside the aspect assembly should be handled as well. We use the term *global references* for describing these references. However, global references pose a problem as they cannot always be evaluated as unambiguous references. Consider the example shown in figure 9.3.

```
class Aspects
{
  public T Advice<T>()
  {
    Aspects2.Foo();
  }
}

class Aspects2
{
  public static void Foo()
  {
    ...
  }
}
```

Figure 9.3: An advice method that might result in an ambiguous reference.

The advice method invokes the static method *Foo()* located in class *Aspects2*. Using a pointcut statement, the user might instruct the method *Aspects2.Foo()* to be inserted at multiple locations within the target assembly. This is a problem, as it is impossible to determine which method should be invoked - the method call is ambiguous. This problem does not exist for references to local constructs, as they can always be mapped locally.

For handling this problem, we have defined a global mapper (called *YIIHAW.Weaver.-GlobalMapper<T>*). All constructs introduced wihtin the target assembly are added to this mapper. Given a reference to the aspect construct, the *Lookup()* method of this mapper returns a reference to a target construct and a flag indicating if the construct is ambiguous:

$$AspectConstruct \longrightarrow (TargetConstruct, IsAmbiguous)$$

A construct is considered ambiguous if it is introduced at multiple locations within the target assembly. If a construct is only introduced once, resolving the reference to the construct is no problem. This means that the user is able to introduce constructs within the target assembly and refer to these constructs (either from advice methods or from other constructs), but only if these constructs are defined once within the target assembly. Constructs that are not referenced anywhere from the aspect assembly can be introduced at as many locations as the user wants.

The implementation of the introduction handling can be found in appendix W. The implementation of the mappers can be found in appendix X.

9.2 Modifying the typestructure

The user can modify the typestructure of the target classes in two ways: By changing the basetype or by implementing one or more interfaces. For either case, the action that should be performed are very similar.

9.2.1 Verifying the availability of the basetypes and interfaces

The weaver checks that new basetypes or new interfaces are available in the target assembly. This means that the user must explicitly introduce these classes into the target assembly himself (using the pointcut specification). YIIHAW does not introduce any constructs automatically, as we believe that the user should be in absolute control of the weaving process.

Typestructure modifications are always applied after all constructs have been introduced into the target assembly. This means that checking whether the new classes or interfaces are available in the target assembly is fairly simple, as the global mapper can be used for this purpose. If a class or interface cannot be found an error is shown and the weaving is aborted.

9.2.2 Checking the definition of methods and properties

Instructing a class to implement an interface makes no sense if the class does not implement the methods, properties and events defined in the interface. YIIHAW checks that all methods and properties of the interface are implemented in the target classes. A similar check is performed for abstract methods when setting a new basetype. If a target class does not implement all the necessary methods and properties the weaving is aborted.

9.2.3 Updating references

When setting a new basetype the weaver needs to update references to the old basetype. Consider the following example:

```
public class BaseClass
{
  ...
}
```

```
public class TargetClass : BaseClass
{
  public TargetClass() : base() // default constructor
  {
  }
}
```

A class, *TargetClass*, is defined which inherits the class *BaseClass*. The default constructor of *TargetClass* would look something like this:

```
ldarg.0
call       instance void BaseClass::.ctor
ret
```

The constructor invokes the base constructor (on class *BaseClass*). This poses a problem when setting a new basetype, as the constructor should no longer refer to *BaseClass*.

To make sure that all references to the basetype of a class are valid, the weaver updates all references to the old basetype in methods, properties, etc.

The implementation for typestructure modifications can be found in appendix X.

9.3 Handling interceptions

Interceptions are the most interesting and at the same time the most complex feature of an aspect weaver. Throughout this report we have discussed various ideas of how to handle interceptions. Based on these ideas, the following section describes the actual implementation of the interception handling.

9.3.1 The join point API

As the user should be able to interact with the weaver from the advice methods, a special join point API need to be defined. This API should allow access to various join point information, as described in section 3.4.2. We have defined this information using a number of properties and methods (defined in the class *YIIHAW.API.JoinPointContext* - see appendix S). The properties and methods are listed in figure 9.4.

Property/method	**Type**	**Action**
AccessSpecifier	string	*ldstr <access specifier for the target method>*
Arguments	string	*ldstr <comma-separated list of arguments>*
DeclaringType	string	*ldstr <declaring type of the intercepted object>*
GetTarget<T>()	T	*ldarg.0* (this-pointer)
IsStatic	bool	*ldc.i4.0* or *ldc.i4.1*
Name	string	*ldstr <name of target method>*
Proceed<T>()	T	inject original target method body
ReturnType	string	*ldstr <return type>*

Figure 9.4: The join point API. The column 'Type' specifies the type of the property/method in the API. The column 'Action' specifes the action taken by YIIHAW when handling this property/method.

Using these properties and methods the user has access to various information about the method or object that is being intercepted. The example below shows how a simple logging advice can be implemented.

```
public T Advice<T>()
{
  Console.WriteLine("entering method: " + JoinPointContext.DeclaringType + ":" +
      JoinPointContext.Name);
  return JoinPointContext.Proceed<T>(); // invoke the original method and return
      the same result
}
```

Calling an intercepted method would produce output like the following:

```
entering method: TargetNamespace.TargetClass:TargetMethod
```

As YIIHAW should add no runtime overhead, all use of these properties are determined statically and replaced with the proper action. For instance, the *DeclaringType* property is replaced with a *ldstr* instruction that specifies the declaring type of the method being intercepted. Thus, the calls to these properties only exist in the advice methods - they are not included in the generated assembly and do therefore not cause any unnecessary overhead.

As it makes no sense to invoke the properties or methods from the *JoinPointContext* outside their intended context (i.e. outside an advice method) the default implementation of these constructs simply throw an exception. That way, the constructs cannot be misused in regular methods (non-advice methods). As these constructs are always replaced at weave-time, there is no problem in using them from an advice method.

9.3.2 Merging the targets and advice

Merging the targets and advice is performed as the last step of the weaving process. This way, all constructs of the aspect assembly that are subject for insertion are available in the mapping tables. This is necessary, as the advice methods might refer to some of these constructs.

The weaver applies the advice to one target method at a time. Multiple advice may be applied to the same target method, if specified so by the user (via the pointcut). The rest of this section describes the approach used for intercepting a single target method. This approach is repeated for each interception statement and for each target method. For the concrete implementation, refer to class *YIIHAW.Weaver.Interception* in appendix X.

Preparing the original target method for insertion

Prior to performing any merging of the advice and the target method a copy of all instructions of the target method is created. For the sake of disussion, we will refer to this copy as *original body* throughout the rest of this section. Recall from previous discussions that the user is not required to keep the original implementation of the method being intercepted: If the user does not invoke the *Proceed* method at any time, this means that he wishes to completely ignore the original implementation. The weaver should thus not operate on the assumption that the original implementation should be maintained. Creating a copy of the body of the target method allows us to subsequently delete all instructions of the target method. This way, the instructions of the advice method can just be copied one by one to the target method without worrying whether they fit into any existing method body. Whenever a call to the *Proceed* method is made

in the advice method, the weaver simply copies all instructions from the *original body* into the target method. This will be elaborated upon later in this section.

Before the *original body* can be inserted into the target method, a couple of things need to be performed. Recall from section 6.2.2 that return statements should be handled with care: Any *ret* instructions in the *original body* would cause the target method to return prematurely. In order to handle this, the weaver inserts a *nop* instruction at the end of the *original body*. It then scans all instructions of the *original body* and replaces all *ret* instruction with an unconditional branch to this *nop* instruction. Branching to a *nop* instruction is necessary at this point, as the last instruction of the *original body* might be a *ret* instruction.

Another thing that needs to be performed on the *original body* is updating all references to local variables. Locals variables of the advice method are always inserted into the target methods, which means that all references in the *original body* will probably refer to the wrong index value. The weaver scans all instructions of the *original body* and updates the index value for all instructions that refer to local variables.

Transferring instructions

Once the *original body* has been updated the actual weaving process begins. The weaver runs sequentially through all instructions found in the advice method and copies each instruction to the target method. If the instruction contain a reference to a construct that has been introduced at a previous stage, the weaver consults the local mapper and the global mapper (defined in section 9.1.2) for translating these references into their proper context.

As some instructions are replaced by others (such as the *ret* instructions of the *original body*, which get replaced with a branch instruction as shown earlier), a mapping-table needs to be maintained for instructions as well. The weaver defines a special class, *RecursiveDictionary<T>* (fully qualified name: *YIIHAW.Weaver.RecursiveDictionary<T>*), for this purpose. Given a reference to an instruction the mapper returns a reference to its substitute:

$OldInstruction \longrightarrow NewInstruction$ (recursive)

As the substitute itself might be replaced by another instruction the mapper is made recursive. The *RecursiveDictionary* class can be seen in appendix X.

Handling Proceed

If an instruction is a method call to the *Proceed* method, special action needs to be taken. Invoking *Proceed* is actually a cue to YIIHAW that specifies that the *original body* should be inserted at this point. As the *original body* has already been prepared for insertion into the target method at a previous stage, the weaver simply copies all instructions into the target body. However, handling calls to *Proceed* is actually a bit more complicated than that, as the type of the variable storing the result of the call to *Proceed* needs to be updated at this point as well if a generic type is used. Recall from section 6.2 that the type of the variable storing the return value should be substituted for the actual type of the method being intercepted. The weaver updates the type of this variable as well as any other variable that refer to this variable (or removes them if the target method is of type *void*). For instance, this is necessary when weaving assemblies compiled in debug-mode, as some of the Microsoft .NET compilers often generate an extra variable (i.e. a variable not explicitly defined in the source code) for storing and retrieving

the return value when compiling in this mode. These variables should be updated as well.

As the type of a generic advice method is replaced with the actual return type of the method being intercepted, YIIHAW does not allow using this type for anything but invoking *Proceed<T>()* or *default(T)*. If the advice method make any other use of the generic type, YIIHAW will abort the weaving process and show an error, as it makes no sense to use the generic type outside the advice methods.

Updating references

When all instructions have been transferred to the target method, the weaver scans all of these instructions, looking for dangling code addresses and unoptimized instructions. A dangling code address might occur if an instruction refers to another instruction that has been removed. For instance, instructions that load or store the return value are either modified or removed by the weaver, as described previously. If a reference exist to such an instruction it will be invalid at this point. The weaver updates all such references using the mapping-table for instructions. Similarily, the weaver check each instruction in order to see if modifying it to a short-form is possible (such as modifying *ldloc* to *ldloc.s*).

Handling try, catch and finally

Try, catch and finally handlers are added to the target method when all instructions have been validated. In Cecil, these handlers are represented as instances of the class *Mono.Cecil.Cil.ExceptionHandler*. Each instance contains a reference to the first and the last instruction for which the handler applies. As some instructions might have been removed or modified, the weaver use the mapping-table for instructions when adding these handlers in order to make sure that the handlers apply to the correct instructions.

9.4 Using YIIHAW

See appendix A for details on how to use YIIHAW.

Chapter 10

Partial functional testing

Generating valid assemblies is a fundamental requirement for any aspect weaver: If the weaver generates assemblies that cannot be executed due to invalid metadata or inconsistent type declarations there is no reason for using an aspect weaver in the first place. In this chapter we will examine the assemblies generated by YIIHAW in order to determine if they contain any form of abnormalities that will cause them to be unverifiable by the CLR.

10.1 The test framework

Even though YIIHAW has generated an assembly and a look through the generated code via *ildasm* shows that the code has been altered as expected, it does not mean that YIIHAW will generate valid assemblies for all cases. To verify the correctness of the assemblies, we have built a test framework and a number of testcases that takes care of validating the assemblies. The source code for the framework and the tests can be seen in appendix R.

When invoking the test framework, an assembly name can be specified. The framework invokes all methods in this assembly that are annotated using the *TestableMethod* attribute and whose declaring class are annotated using the *TestableClass* attribute. A test method can use the framework API to perform various tasks, such as creating a pointcut file and invoking the weaver. When invoking the weaver, the test framework captures the output generated by YIIHAW and checks if the weaving was successful. This is done by scanning the output text from YIIHAW and looking for any indications that an error occurred. Sometimes it is useful to be able to check that a given exception was thrown by the weaver. The framework API supports defining the expected output from YIIHAW. This is compared to the actual output and an error is reported if they differ.

To verify that the assembly behaves as expected, the framework API can execute a method in the generated assembly and compare the returned value with an expected value. Even though this kind of test is very simple, we believe it to be valuable, as some well considered tests can make up for this simplicity.

For validating the assemblies, the framework uses the PEVerify tool that is included in the .NET framework. The framework invokes PEVerify on the generated assembly after executing the test method. If PEVerify returns an error this is captured by the test framework.

When all test methods have been executed and the generated assemblies have been verified, all errors that were encountered are printed to the console.

10.2 The testcases

The testcases that we have defined are designed to test a wide range of the functionality provided by YIIHAW. Of course the tests covers the three main types of actions supported by YIIHAW: Introductions, interceptions and modifications. For interceptions, the main focus of the tests have been on the different possibilities regarding the return types of the advice methods, such as specific types, *void*, and generic types. Furthermore, the tests make use of all properties and methods found in the YIIHAW join point API.

The pointcut language is also subject for various tests that cover the different possibilities for each type of statement. We have created tests where the aspects contain opcodes that we know can be troublesome for the weaver to handle, such as opcodes that refer to types, methods, fields, etc. and opcodes that refer to variables and arguments. A full scheme for the tests can be seen in appendix Q.

10.2.1 A test sample

In figure 10.1 one of the tests are depicted. Looking at the test method *Test2()*, it can be seen that it is annotated using the *TestableMethod* attribute. On this attribute, the target file is specified. This is used by the framework when it performs the verification of the target file. The method starts by instructing the test framework to create a new pointcut file. Afterwards the test framework is instructed to perform the actual weaving of the target assembly using YIIHAW. In the third and fourth statements the arguments to the *ExpectedReturnOnCall()* method are created, which are used for comparing the return value of a given method with the expected value. These arguments need to be passed as an object array, as the test framework uses reflection to call the method. In this specific example the expected value is "Hello Hello".

```
//The advice method
public string AdviceTest2(string a)
{
    return JoinPointContext.Proceed<string>() + " " + a;
}

//The target method
public string TargetTest2(string a)
{
    return a;
}

//The test method
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test2()
{
    API.CreatePointcutFile("around public * * *.target.Target:TargetTest2(string)
        do Aspect.Aspect:AdviceTest2;", "pointcutfile");
    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect/bin/
        Release/Aspect.dll", "pointcutfile");
    object[] args = new object[1];
    args[0] = "Hello";
    API.ExpectReturnOnCall("Hello Hello", "Target", "Test2", args);
}
```

Figure 10.1: An example of a testcase, showing the advice, target, and the actual test method which includes calls to the *CreatePointcutFile* and *ExpectReturnOnCall* methods of the test API.

Chapter 11

Measuring the runtime performance

The overall objective of YIIHAW is to apply aspects in an efficient manner, i.e. the runtime overhead on the woven code should be kept at a minimum. For this reason, it is interesting to see how YIIHAW performs compared to other aspect weavers and handwritten code. This chapter examines the runtime performance of the generated assemblies by applying various aspects to a number of target assemblies. We will evaluate on these results in the next chapter.

11.1 Test setup

The execution time is measured in milliseconds as an average of 50 testruns for all tests. The average deviation in execution time is calculated as well. All tests are compared to an implementation coded by hand, which can be considered as the optimal implementation.

All tests are performed on a machine with the same specifications as in section 2.3. All tests are compiled as *release builds* with *code optimization* turned on.

11.2 Comparing YIIHAW to code written by hand

Section 2.3 examined the runtime performance penalties incurred when using some of the existing aspect weavers. This was done on the base of seven small tests that had the purpose of measuring the mean effect of applying various aspects to a target assembly. In order to measure the efficiency of our weaver it makes sense to implement these tests in YIIHAW as well. The results of these tests can be seen in figure 11.1. The source code and the pointcuts for the implementation of these tests in YIIHAW can be seen in appendix K.

Recall that YIIHAW does not support the use of *before* and *after interception*. However, as these can be simulated using *around interception*, Test 2 and 3 are identical to Test 1 when implemented in YIIHAW (and when coded by hand as well). For this reason, the results of Test 2 and 3 are not shown in the test results.

Even though the implementation coded by hand have already been tested once (in chapter 2), all tests have been performed once again in order to minimize any uncertainty factors.

11.3 Implementing a generator for a collection library

The test described above are all somewhat "synthetical": They are very simple, as they only measure one single action at a time, such as intercepting or introducing a single method. As

Aspect weaver	Test 1	Test 4	Test 5	Test 6	Test 7
Coded by hand	718 (1.00) 0.36%	717 (1.00) 0.15%	720 (1.00) 0.20%	605 (1.00) 0.36%	744 (1.00) 0.59%
YIIHAW	717 (1.00) 0.32%	716 (1.00) 0.15%	720 (1.00) 0.22%	605 (1.00) 0.30%	744 (1.00) 0.36%

Figure 11.1: Test results. The first number in each cell is the average execution time in milliseconds. The second number (in parentheses) specifies the execution time as a factor of the reference implementation. The third number is the average deviation (in percent) from the average execution time for each testrun.

described throughout this report, the primary motivation for building YIIHAW was to create an efficient aspect weaver that can be used for the generation of specialized programs. In order to provide a more "real life usage" of the weaver, we will perform some tests that are somewhat more complex, as they apply various constructs that are intertwined within eachother. This means that YIIHAW must be able to resolve cross-references between various constructs and be able to efficiently merge the aspects into the target assembly. The tests will be based upon the same tests that we made in the project "Generation of specialized collection libraries" [3].

11.3.1 Test scenario

A target assembly is defined, which contain two collection classes, *ArrayList* and *LinkedList*. The *ArrayList* class uses an array for storing elements. The *LinkedList* class uses a linked list for storing elements. Using an aspect weaver, support for events and enumeration should be added to the assembly.

Supporting events requires that:

- An event (of type *System.EventHandler*) is introduced into both classes.
- A method (named *OnChanged*) is introduced into both classes. This method uses the event that is inserted.
- Methods that update the collections (such as the *Add* and *Remove* methods) should be intercepted, making them invoke the *OnChanged* method.

Supporting enumeration requires that:

- Two interfaces, *IEnumerator* and *IEnumerable*, are introduced into the assembly.
- A field (named *stamp*) is introduced into both classes.
- An enumerator class is introduced as a nested class into both classes.
- Both classes implement the interface *IEnumerable* and the corresponding method, *GetEnumerator*, is introduced.
- Methods that update the collections should be intercepted, making them update the inserted field *stamp*.

- The *Equals* method of the *ArrayList* should be completely replaced with a new implementation that uses an enumerator for looping through items in the *LinkedList* class.

An ad hoc program is used for performing the tests. This program performs a number of operations, such as adding and removing a fixed number of items to the collections, enumerating the collections and comparing the collections. The program make use of all constructs that are modified or introduced by the aspect weaver. The source code for the test program can be seen in appendix P.

11.3.2 Test results

The results of the tests are shown below.

Aspect weaver	Events	Enumerations
Coded by hand	8547 (1.00) 0.42%	602 (1.00) 0.36
YIIHAW	8545 (1.00) 0.28%	600 (1.00) 0.36%
AspectDNG	13941 (1.63) 0.09%	30247 (50.2) 0.53%

Figure 11.2: Test results. The first number in each cell is the average execution time in milliseconds. The second number (in parentheses) specifies the execution time as a factor of the reference implementation. The third number is the average deviation (in percent) from the average execution time for each testrun.

AspectDNG is included in the test results, as this is the aspect weaver that were used during the original tests. The results for AspectDNG are only included in order to show the effect of applying the aspects using another weaver than YIIHAW. When evaluating the performance of YIIHAW in the next chapter, we will compare the results to the implementation coded by hand, as this yields the most interesting comparison.

The source code and pointcuts for AspectDNG, YIIHAW and the implementation coded by hand can be seen in appendix L - N. The source code for the target assembly can be seen in appendix O.

Chapter 12

Evaluation

For all tests defined in appendix R the test framework returns no errors or warnings. This means that all assemblies generated by YIIHAW in these tests are valid according to PEVerify and returns whatever value was expected when invoking the applied aspects. Even though these tests do not provide an exhaustive check of all possible combinations of input (pointcut, target and aspects), we are confident that they cover the most important and critical parts related to generating an assembly.

As mentioned in section 10.2 some of the tests focus on checking instructions that contain operands referring to types, methods, fields, etc. These operands are interesting, as it is essential that they are mapped correctly by YIIHAW in order to generate valid output. The tests do not cover every single use of such operands (as it is practically impossible to test all of them), but only a representative subset of them. However, during the implementation of YIIHAW we have checked that every single type of operand that refers to other constructs[1] are explicitly handled by the weaver. We do therefore not expect that any operand are left unchecked by YIIHAW.

12.1 Runtime performance of the generated assemblies

Looking at the test results in section 11.2 the overall conclusion is clear: For all cases, the assemblies generated by YIIHAW is identical to the implementation coded by hand in terms of runtime efficiency. In some cases, the generated assemblies are actually a bit faster than the reference implementation, although this cannot be generalized as the differences are simply too small (1 millisecond). Looking at the assemblies via *ildasm* shows that all methods intercepted are directly comparable to that of the reference implementation - the only difference is the addition of a single *nop* instruction just before the last *ret* instruction. This *nop* is added due to the way return statements of the original target method are handled by YIIHAW (refer to the discussion of this issue in section 6.2.2 and 9.3.2). However, as *nop* instructions use practically no CPU-time, they do not have any effect on the results. Adding methods (Test 6) and changing the superclass (Test 7) yields exactly the same output as the reference implementation and does therefore obviously not introduce any overhead.

The tests described in section 11.3 provide a more ”real-life” usage of an aspect weaver, as they apply multiple, mutually dependent aspects at once. These tests are thus somewhat more complex to handle by the weaver, as it must be capable of mapping these aspects into the new context without generating any runtime overhead. Looking at the test results it can be seen

[1]These operands were identified by consulting the total list of opcodes found in the Common Langugage Specification [2].

that the execution time of the assembly generated by YIIHAW is comparable to the implementation coded by hand: For both tests the generated assembly are two milliseconds faster than the reference implementation. Again, this cannot be generalized, as the difference is too small.

Examining the generated assemblies using *ildasm*, it can be seen that the only overhead added by YIIHAW is the loading and storing of the return values and possibly a single *nop* instruction. Figure 12.1 shows an example of this: The original *Add(int,object)* method found in the target assembly returns the value *true* (which is represented as a *ldc.i4.1* instruction in CIL). Adding support for events requires that the *OnChanged()* method is invoked at the end of the method. This means that the advice method[2] invokes the original method (using *Proceed()*), stores the result in a variable, invokes the *OnChanged()* method and finally returns the original return value (which was previously stored in a variable). Comparing the instructions of the reference implementation and the generated assembly in figure 12.1 it can be seen that the only difference is the additional *nop* instruction (instruction no. 7), the storing of the return value (instruction no. 8) and the loading of the return value when returning (instruction no. 12). Besides these three instructions the implementations are completely identical. This result applies to all other methods that were intercepted as well. We consider this to be the optimal output, as avoiding these extra instructions would be nearly impossible: It would require very complex analysis and modifications of the target and advice method by the weaver, as all IL-instructions would have to be restructured in order to get the right return value on the stack before returning. We consider such an approach to be too advanced compared to the very small performance improvement that might be achieved.

All constructs introduced by YIIHAW are identical to those found in the reference implementation. Similarily, modifying the typestructure does not yield any difference compared to the reference implementation.

```
YIIHAW:                         Coded by hand:
-------------------------------------------
...                             ...
dup                             dup
ldfld                           ldfld
ldc.i4.1                        ldc.i4.1
add                             add
stfld                           stfld
ldc.i4.1                        ldarg.0
nop                             ldsfld
stloc.0                         callvirt
ldarg.0                         ldc.i4.1
ldsfld                          ret
callvirt
ldloc.0
ret
```

Figure 12.1: The bottommost instructions of method *Add(int,object)* of class *ArrayList*. To the left can be seen the the output generated by YIIHAW. To the right can be seen the implementation coded by hand. The *callvirt* instruction found at the end invokes the *OnChanged()* method. Operands are left out, as they are not relevant for this discussion.

[2]Refer to appendix M for the concrete implementation of this advice method.

The poor results achieved by AspectDNG is due to the use of reflection: AspectDNG does not map references to constructs in the aspect assembly. This means that the only way to refer to such constructs is by using reflection, which obviously has a significant runtime overhead.

12.2 The aspect language

There are significant differences when comparing the implementation of the aspects in YIIHAW and AspectDNG. YIIHAWs mapping of local and global constructs greatly simplifies the code that needs to be written and provides a much more typesafe approach than AspectDNG. Figure 12.2 shows the aspect code for implementing support for events in AspectDNG. Figure 12.3 shows the corresponding implementation in YIIHAW. The *OnChanged()* method shown in these examples refer to the field *changed* (which should be inserted into the target assembly along with the *OnChanged()* method). As AspectDNG does not perform any mapping of such references, the use of reflection is needed for accessing this field. This has a huge impact on the runtime overhead. Furthermore, you loose the typesafety normally ensured by the compiler.

```
class OnChangedMethodsLinkedList
{
    public event EventHandler changed;

    ...

    public void OnChanged(System.EventArgs e)
    {
        object _this = this;
        FieldInfo field = ((Collections.LinkedList)_this).GetType().GetField("
            changed", BindingFlags.Public | BindingFlags.Instance | BindingFlags.
            NonPublic); // reflectively get the "Changed" eventhandler

        if (field != null)
        {
            EventHandler handler = (EventHandler)field.GetValue(this);  // cast
                the "Changed" field to a System.EventHandler
            if (handler != null)
                handler(this, e);
        }
    }
}
```

Figure 12.2: Implementation of the event aspect in AspectDNG.

For methods, properties and events a better approach can be used in AspectDNG: Define a wrapper-interface instead and typecast to this interface whenever you need to access such constructs[3]. However, this is still a very messy approach, as you add interfaces to the target assembly that are only needed to please AspectDNG - they make no sense once the aspects have been applied. Where possible, we have used this approach when implementing the aspects in AspectDNG, as it does not incur as much overhead as reflection.

The implementation of the event aspect is much simpler in YIIHAW: There is no need for wrapper-interfaces or reflection, as all references are automatically mapped into the new context. The user still needs to provide the pointcut for introducing the referenced constructs within the

[3]This is not possible with fields, as these cannot be declared in an interface.

```
class EventConstructs
{
    public event EventHandler changed;

    ...

    public void OnChanged(System.EventArgs e)
    {
        if (changed != null)
            changed(this, e);
    }
}
```

Figure 12.3: Implementation of the event aspect in YIIHAW.

target assembly, but this task is fairly simple and does not require any special means in the implementation. We believe that the approach used by YIIHAW provides a much more simple, understandable and flexible approach than that of AspectDNG.

For a complete implementation of the aspects in AspectDNG and YIIHAW, refer to appendix L and M.

Chapter 13

Future work

Although the current implementation of YIIHAW behaves and performs as expected there are still a few issues that needs to be resolved in a future release.

13.1 Support for further introductions

Currently the weaver support introducing classes, methods, fields, properties and events. It should be possible to introduce delegates, enumerations and attributes as well. The pointcut language already support these kind of introductions, but they are not handled by the weaver.

13.2 Handling typestructure modifications

The checks performed by the weaver when handling basetype modifcations are not thorough enough for all cases. YIIHAW updates all references to constructs in the old basetype and make sure that they refer to the corresponding constructs in the new basetype. However, when handling methods and properties the current implementation only check that the name of the method or property match those declared in the old basetype. There are no checks on the return type or type of arguments.

Similarily, the weaver should check that existing constructs found in the target class will work when changing the basetype. For instance, methods found in the target class might assume that the current class implements a certain interface - if the new basetype does not implement this interface the assembly might be invalid. We expect this kind of check to be relatively complicated to implement, as there are a lot of issues that need to be resolved in this regard.

13.3 Further possibilities in the join point context API

Most of the properties in the *YIIHAW.API.JoinPointContext* class are of type *string*. It is possible to provide greater possibilities when using this API, as it can be expanded so that some of the information can be returned as other types as well. For instance, returning a *Type* object representing the return type of the target method would be useful, as that would allow the user to introspect the types using reflection.

13.4 Compatibility with older releases of .NET

YIIHAW has only been tested using .NET 2.0. It is possible that the weaver has trouble weaving assemblies compiled using an older version of the .NET compiler. We do not expect this to be

the case though, as previous releases of .NET all contain the same subset of opcodes as .NET 2.0. However, this need to be tested in order to be sure.

13.5 Accessing generic parameters

The join point API offers various ways of getting information about the method currently being intercepted. However, the current implementation does not support fetching any generic parameters on these methods. These should be accessible along with any other types of parameters. We expect this to be a relatively simple task.

13.6 Support for generics in the pointcut language

YIIHAW support intercepting and introducing generic methods and classes. However, the pointcut language does at this point not support generics. This means that you cannot specify any of the generic parameters of the construct you wish to target in the pointcut file. Thus, generic constructs can only be matched by other properties (such as the name of the method, return type, etc.).

13.7 YIIHAW on the web

YIIHAW is created as a project at http://sourceforge.net/projects/yiihaw. Any future release will be announced on this site.

Chapter 14

Conclusion

Throughout this thesis we have examined various aspect weavers that exist for the .NET platform with the purpose of determining their feasibility for weaving performance-critical applications. Our main criteria in this regard was runtime efficiency: When measuring the runtime performance, the generated programs should be comparable to similar implementations coded by hand. None of the aspect weavers fulfilled this requirement, although Aspect.NET achieved fairly good results. However, Aspect.NET lacks many of the basic features needed when using AOP, such as introductions and intercepting instance methods. Furthermore, Aspect.NET's handling of interceptions is somewhat primitive, which causes invalid assemblies to be generated for some scenarios. We thus consider neither of the aspect weavers that were examined to be usable for applying aspects to performance-critical applications (see section 2.4).

In order to provide an effective solution for handling these kind of applications, we have implemented our own aspect weaver that uses an inlining-approach when handling interceptions. This approach turned out to be very useful, as it avoids many of the common problems found in other aspect weavers: Our weaver introduce no unnecessary constructs into the target assemblies, which means that the program structure defined by the user is completely maintained once the aspects have been applied. Furthermore, as all advice gets transferred into the target assembly during weaving, this means that the generated assemblies will be completely self-contained once the weaving has been performed - no extra assembly dependencies are introduced.

The implemented prototype yields impressive results in terms of runtime efficiency: For all tests, the performance of the generated assemblies directly match similar implementations coded by hand (refer to chapter 11). These tests measured the mean effect of applying various aspects to precompiled target assemblies. Inspecting the generated assemblies, it can be seen that the only overhead caused by the weaver is the introduction of a few *nop* instructions and possibly some load and store instructions (*ldloc* and *stloc*). Considering the nature of the test scenarios, we consider this to be the optimal output for an aspect weaver.

We believe that the proposed aspect language provides a very flexible, intuitive and secure interface for the weaver that makes it fairly easy to implement the constructs needed. For instance, the use of introductions does not require any special preparations of the constructs that should be introduced - any construct can be inserted into the target assembly, regardless of how it is defined in the aspect assembly. Similarily, the syntax used for interceptions ensures that only typesafe implementations are created that directly prevents the user from returning types that are incompatible with the target methods. Furthermore, the weaver typechecks and translates all references within the aspect assemblies making it practically impossible to generate invalid assemblies. This is confirmed by the tests described in chapter 10.

The weaver support the three main AOP-features: Interceptions, introductions and type-structure modifications, thereby supporting all of the features required for implementing program-generators, such as the one described in our previous project: "Generation of specialized collection libraries" [3]. Even though the weaver does not currently support introducing all types of constructs (refer to the issues discussed in the previous chapter), we do consider it to be highly usable for implementing efficient program-generators.

Index

.NET attributes, 38

abstract syntax tree, 27
advice, 9
advice inlining, 29, 48
advice method, 9
advice syntax, 48
after interception, 10
annotation, 38
around body, 9
around call, 9
around interception, 9
aspect language, 83
aspect weaver, 7
Aspect.NET, 12, 21
AspectC++, 26
AspectDNG, 12, 83
assembly, 41

before interception, 10
binding mode, 25

call interceptions, 21
Cecil, 47
cflow, 9
checking references, 53
CIL, 41
CIL datatypes, 43
code scattering, 11
code tangling, 11
CodeDom, 27
Common Intermediate Language, 41
Common Language Runtime, 41
cross-cutting concern, 7

debug file, 26
direct advice invocation, 28
dynamic pointcuts, 9
dynamic residue, 9
dynamic weaving, 25

exception handling, 43

flow control, 42

generator, 78
GetTarget, 57
global mapping, 70
global reference, 70

IL Assembler, 41, 46
IL Disassembler, 46
instead, 21
instruction, 42
intercepted target, 48
interception, 9, 72
introduction, 10, 68

join point, 8, 72
join point context, 8

local mapping, 70
local reference, 70

merging local variables, 53
metadata, 41
module, 41
mutual dependent reference, 69

obliviousness, 8
opcodes, 18
operand, 42

parsing method, 64
PDB file, 26
Phoenix, 46
pointcut, 8, 38, 59
pointcut grammar, 61
pointcut language, 61
pointcut parser, 64
pointcut specification, 59
pointcut syntax, 61
preprocessing, 26
proceed, 9, 50, 74

quantification, 8

Rapier LOOM, 13
recursive dictionary, 74
runtime performance, 78

scanner, 63
separation of concerns, 7
source code weaving, 26
stack, 43
static evaluation, 12
static weaving, 25

target code, 8
target method, 10
test, 77
test framework, 76
token, 63
tokenizer, 63
two-pass introduction, 69
typestructure modification, 10, 71

weaving, 68
wildcard, 39
wrapper interfaces, 83

Bibliography

[1] Jason Bock: "CIL Programming: Under the hood of .NET", Apress, 1st edition, 2002, ISBN: 1590590414

[2] Serge Lidin: "Expert .NET 2.0 IL Assembler", Apress, 1st edition, 2006, ISBN: 1590596463

[3] Rasmus Johansen & Stephan Spangenberg: "Generation of specialized collection libraries", 2006, http://itu.dk/people/spangenberg/main.pdf

[4] The C5 Generic Collection Library for C# and CLI, 2005, http://www.itu.dk/research/c5/

[5] AspectDNG, http://www.dotnetguru.org/sarl/aspectdng/

[6] AspectJ, http://www.eclipse.org/aspectj/

[7] Daniel Lohmann et al: "A quantitative analysis of aspects in the eCos kernel", 2006

[8] Tzilla Elrad, Robert E. Filman & Atef Bader: "Aspect-oriented programming", 2001

[9] Bruno Dufour et al: "Measuring the dynamic behaviour of AspectJ programs", 2004

[10] Rapier LOOM, http://www.dcl.hpi.uni-potsdam.de/research/loom/

[11] Wolfgang Schult, Peter Troeger and Andreas Polze: "LOOM .NET- An Aspect Weaving Tool", 2003

[12] AspectC++, http://www.aspectc.org/

[13] Edsger W. Dijkstra: "EWD 447: On the role of scientific thought", 1974

[14] Peter Sestoft: "Grammars and parsing with Java", 1999, http://www.dina.kvl.dk/~sestoft/programmering/parsernotes.pdf

[15] Aspect.NET, http://www.msdnaacr.net/curriculum/pfv.aspx?ID=6595

[16] The Mono Project, http://www.mono-project.com/

[17] NKalore, http://aspectsharpcomp.sourceforge.net/

[18] Microsoft Phoenix ®, http://research.microsoft.com/phoenix

[19] Howard Kim, "AspectC#: An AOSD implementation for C#.", master thesis at Department Of Computer Science Trinity College Dublin

[20] Cecil, http://www.mono-project.com/Cecil

[21] Personal communication with Vladimir O. Safonov, <04-11-2006>.

[22] Microsoft Phoenix C++ compiler backend C2, http://research.microsoft.com/phoenix/compiler.aspx

Appendix A

Usage guide for YIIHAW

(This is the same guide as found on the YIIHAW homepage: http://sourceforge.net/projects/yiihaw)

Invoking YIIHAW

YIIHAW is implemented as a simple command-line program. The general syntax is defined like this:

```
yiihaw <pointcut file> <target assembly> <aspect assembly> [output assembly] [-v]
```

Properties written in angle brackets are mandatory. Properties written in square brackets are optional.

If no output assembly is specified, the name of the target assembly is used (thereby overwriting the target assembly). Using the optional "-v" argument puts YIIHAW in verbose mode, which means that detailed information regarding the weaving is shown.

Introducing constructs

YIIHAW currently support introducing methods, properties, classes, fields and events. There are no restrictions on their type or how they are defined - any of these types of constructs defined in the aspect assembly can be introduced. YIIHAW insert the constructs exactly as it is defined within the aspect assembly. For instance, if you define a method as being "public static void" it will remain so in the target assembly. It is not possible to instruct the weaver to insert a private method and make it public in the target assembly.

For details about how to instruct YIIHAW to introduce constructs, see the pointcut language.

Typestructure modification

Using YIIHAW you can make two types of typestructure modifications:

- change the basetype of one or more classes
- implement one or more interfaces

You can instruct a class to implement as many interfaces as you want. YIIHAW will check that the target classes implement all the methods, properties and events of the interfaces. If some of these constructs are not located in the target class already, you need to instruct YIIHAW to insert them first (using the pointcut language).

For details about how to instruct YIIHAW to make these modifications, see the pointcut language.

Intercepting methods

YIIHAW can intercept any kind of method. A typical advice method might look like this:

```
public class Aspects
{
  public int Advice(string s)
  {
    Console.WriteLine("value of s is: " + s);
    return YIIHAW.API.JoinPointContext.Proceed<int>();
  }
}
```

This advice method can be used for intercepting any method in the target assembly that returns *int* and takes a string as the first argument. Thus, the following target methods can be intercepted using this advice method:

```
public int TargetMethodA(string x)
{
  ...
}

public int TargetMethodB(string s, int i, float f)
{
  ...
}
```

Notice that the name of the arguments do not need to match the exact name given in the advice method - only it's type must match (”string” in this example). Similarily, the target methods do not need to match the number of arguments of the advice method: As long as the target method matches all arguments of the advice method (in the same order as defined in the advice method), it doesn't matter how many arguments the target methods contain. Using arguments of the method being intercepted is as simple as using the arguments defined on the advice methods. Use of these arguments are automatically mapped by YIIHAW to match the intended argument in the target methods.

The original target method can be invoked using the Proceed<T>() method defined as a static method on the YIIHAW.API.JoinPointContext class. You need to add a reference to the YIIHAW.API DLL in order to invoke this method. The Proceed method takes a single generic argument, which specifies the type of object being returned. You should always specify the same type as the return type of the advice method (”int” in the example shown above). Using generics allows a more typesafe approach, as this allows the compiler to typecheck the advice method.

Implementing ”catch-all” advice methods

The advice method shown above can only be used for intercepting target methods that return an int. Sometimes you need to be able to make an advice method that can intercept any kind of method, regardless of its type. This can be done by defining the advice method to return a generic type:

```
public class Aspects
{
  public static T Advice<T>()
  {
    Console.WriteLine("advice method here...");
    return YIIHAW.API.JoinPointContext.Proceed<T>();
  }
}
```

This advice method can be used for intercepting any kind of method. The type T is used as a substitute for the actual return type of the target method being intercepted (the generic parameter does not need to be named T - you can use whatever name you like). YIIHAW automatically replaces T with the actual type when applying this advice method to a target method. Thus, you do not need to consider the actual type being intercepted when using this syntax. Advice methods defined using a generic parameter can take arguments as well. Consider the following advice method:

```
public class Aspects
{
  public static T Advice<T>(int i, string s)
  {
    ...
  }
}
```

This advice method can be used for intercepting any method that takes an int and a string as the first two arguments.

Void methods

The Proceed method always takes a generic argument describing the type of object to return. You should always use the same type as the return type of the advice method. If you define an advice method with return type ”void” this poses a problem, as ”void” is not a valid generic type. For these cases, use the special ”Void” class included in the YIIHAW API:

```
public class Aspects
{
  public static void Advice()
  {
    YIIHAW.API.JoinPointContext.Proceed<YIIHAW.API.Void>();
  }
}
```

This issue only applies to advice methods that are defined as returning ”void”. It does not apply to advice methods of type T (or any other generic parameter), even though such an advice method can be used for interceping target methods of type void.

Invocation kind of the advice method

Advice methods defined as being static can be used for intercepting all methods (both static and instance methods). Advice methods defined as an instance type can only be used for intercepting instance methods. Thus, define the advice method as being static if you need to be able to intercept any target method.

Storing the result of Proceed

You don't need to return the result of Proceed immediatly. You can store the result in a variable and return it later (or even return something else):

```
public class Aspects
{
  public static T Advice<T>()
  {
    T result = YIIHAW.API.JoinPointContext.Proceed<T>();
    ...
    return result;
  }
}
```

You can even use the result in whatever way you like:

```
public class Aspects
{
  public static int Advice()
  {
    int result = YIIHAW.API.JoinPointContext.Proceed<int>();
    ...
    return result * 7;
  }
}
```

This advice method invokes the original target methods, takes it return value, multiplies it by 7 and returns the new value (effectively overriding the original return value).

The YIIHAW API

You have already seen how to use the Proceed method. Below can be seen a complete list of all properties and methods defined in the YIIHAW API.

Property/method	**Type**	**Action**
Access specifier	string	Returns the access specifier of the method being intercepted
Arguments	string	Returns a comma-separated list of all arguments of the target method
DeclaringType	string	Returns the name of the declaring type of the method being intercepted
GetTarget<T>()	T	Returns a reference (this-pointer) to the object being intercepted
IsStatic	bool	Returns a boolean value indicating if the target method is static or not
Name	string	Returns the name of the method being intercepted
Proceed<T>()	T	Injects the original target method and returns its value
ReturnType	string	Returns the return type of the method being intercepted

Using this API you can write an advice method that logs all methods that are invoked:

```
using YIIHAW.API;

public class Aspects
{
  public static T Advice<T>()
  {
    Console.WriteLine("entering method: " + JoinPointContext.DeclaringType + ":" +
        JoinPointContext.Name);
    return JoinPointContext.Proceed<T>();
  }
}
```

This advice method would produce output like the following:

```
entering method: TargetNamespace.TargetClass:TargetMethod
```

All of these properties are determined statically and replaced with the proper action. This mean that they do not add any runtime overhead (that is: they are not determined runtime).

Referring to the intercepted object

Using the GetTarget method you can obtain a reference to object that you are intercepting:

```
using YIIHAW.API;

public class Aspects
{
  public static T Advice<T>()
  {
    TargetClass original_target = JoinPointContext.GetTarget<TargetClass>();
    original_target.SomeMethod();
    return JoinPointContext.Proceed<T>();
  }
}
```

This example fetches a reference to the intercepted object (of type ”TargetClass”) and invokes a method on it (assume that TargetClass contains a method called ”SomeMethod”). Obtaining such a reference obviously requires that you add a reference to your target assembly from the aspects assembly before the code will be compilable. Note that the GetTarget method is only valid when intercepting instance methods (it does not make sense to get a reference to the object containing a static method). YIIHAW will check this for you.

The pointcut language

All pointcuts are defined in a separate text-file. There are no restrictions on the name or extension of this file - you can name it whatever you like. YIIHAW defines three types of pointcut statements:

- introductions
- interceptions

- typestructure modifications

The general syntax for these statements are as follows:

Interceptions:

```
around <access> <invocation kind> <return type> <type>:<method(arguments)>
[inherits <type>] do <advice type>:<advice method>;
```

Introductions:

```
insert <construct> <access> <invocation kind> <return type>
<aspect type>:<aspect name[(arguments)]> into <type>;
```

Typestructure modifications:

```
modify <type> <action> <aspect type>;
```

Statements in angle brackets are mandatory. Statements in square brackets are optional. All statements must be terminated with a semi-colon.

Interceptions

An interception statement start with the keyword ”around” (YIIHAW only supports around interception). All properties between ”around” and ”do” describe the methods in the target assembly which should be intercepted. You can use wildcards (*) for all of these properties if you like. The properties following the keyword ”do” describe the advice method(s) to use. Some examples:

```
(a) around public static void TargetNamespace.TargetClass:Foo(int,string)
    do AdviceNamespace.AdviceClass:AdviceMethod;

(b) around * * * *.*:*(*) inherits System.Collections.Hashtable
    do AdviceNamespace.AdviceClass:AdviceMethod;
```

The first statement (a) matches all methods that:

- are public
- are static
- return ”void”
- are defined on the type ”TargetNamespace.TargetClass”
- are named ”Foo”
- takes two arguments: int and string (in that order)

The method named ”AdviceMethod” defined on the type ”AdviceNamespace.AdviceClass” is used as advice when intercepting the target methods.

The second statement (b) matches all methods that:

- has any access specifier (as a ”*” is used)
- has any invocation kind (both static and instance)
- has any return type
- are defined on any type (written ”*.*”)
- has any name
- takes any number of arguments (of any type)
- inherits ”System.Collections.Hashtable” (actually, it is the declaring type of the target method that needs to inherit from this class)

Thus, this statement matches any method whose declaring type inherits ”System.Collections.-Hashtable”. Leaving out the inherits property, the statement would match any method.

Notice that you do not specify the complete signature of the advice method to use - you only specify the name of the method. This means that you can write as many advice methods with the same name as you like (of course, they have to vary by signature, otherwise the code will not be compilable). For each target method that are going to be intercepted, YIIHAW will pick the advice method that has the best match for that target method. Suppose you define the following two advice methods:

```
public class Aspects
{
  public static T Advice<T>()
  {
    Console.WriteLine("catch-all advice method here...");
    return JoinPointContext.Proceed<T>();
  }

  public static int Advice()
  {
    Console.WriteLine("int advice method here...");
    return JoinPointContext.Proceed<int>();
  }
}
```

The advice method of type int would be used for all target methods that return “int”. The generic advice method would be used for all other types of methods. Thus, YIIHAW always pick the advice method that has the closest match with the signature of the target method. The advice method can match the target methods in two ways: By the return type or/and by the method arguments. YIIHAW will always pick the advice method that has the same return type as the target method (if possible) and pick the advice method that has the highest number of arguments in common with the target method. If given the choice of picking an advice method that match all arguments or an advice method that matches the return type, YIIHAW will pick the latter.

Introductions

An introduction start with the keyword ”insert” followed by a property (named ”construct”) defining the type construct to insert. There are five possible values for this property:

- method
- property
- field
- class
- event

All properties preceding the keyword ”into” describe the construct that should be inserted (from the aspect assembly). No wildcards are allowed for these properties. Note that the properties ”access”, ”invocation kind” and ”return type” should not be specified when introducing classes (as these make no sense for classes). They are mandatory for all other constructs.

The insert-statement must match one specific construct - it must not match multiple constructs. The property suceeding the keyword ”into” describe the type (namespace and class) in which the construct should be inserted. Some examples:

```
(c) insert method public * int Namespace.AspectClass:Foo(string,System.Object)
    into TargetNamespace.Class;

(d) insert class Namespace.AspectClass into TargetNamespace;

(e) insert delegate Namespace.AspectClass:MyDelegate(int,string)
    into TargetNamespace.*;
```

The first statement (c) matches the constructs that:

- are a method
- are public
- are of any invocation kind (static and instance)
- return an integer
- are defined on the type ”Namspace.AspectClass”
- are named ”Foo”
- take two arguments (of type string and object)

This construct is inserted into the type ”TargetNamespace.Class”.

Typestructure modification

A typestruction modification start with the keyword ”modify” followed by the type (from the target assembly) which should be modified. The property named ”action” defines what the weaver should do with this type. There are two possible values for this property: ”inherit” or ”implement”. If the former is specified, the weaver will make the target type inherit the aspect type specified. If the latter is specified, the weaver will make the target type implement the aspect type specified (which must be an interface for obvious matters). YIIHAW will check that all methods of the interface are implemented as well. If this is not the case, an error will be shown. Some examples:

```
(f) modify TargetNamespace.Class inherit Namespace.AspectClass;

(g) modify TargetNamespace.Class implement Namespace.AspectInterface;
```

Short form notation for types

All types defined in the statements (for example, the return type of a method) must be fully specified (namespace and class). The only exception to this rule are the standard types defined in .NET, such as *string*, *int*, *float*, etc. These types do not need to be fully specified (although you are allowed to do so).

Appendix B

Pointcut grammar

```
start = around | insert | modify .

around = "around" access membertype returntype name ":"
          method inherit "do" forcedname ":" forcedname .

insert = "insert" introduction .

introduction = typeintro | memberintro .

modify = "modify" forcedname forcedinherit .

typeintro = typeconstruct forcedname "into" name .

typeconstruct = "class" | "attribute" | "enum" -

memberintro = memberconstructwitharg access membertype returntype forcedname ":"
              forcedmethod "into" name
              | memberconstruct access membertype returntype
              forcedname ":" forcedname "into" name .

memberconstruct = "property" | "field" | "event" .

memberconstructwitharg = "method" | "delegate" .

access = "public" | "private" | "protected" | "internal" | "*" .

membertype = "static" | "instance" | "*" .

returntype = name | "void" | "*" .

name = string | "*" .

forcedname = string .

method = name "(" arglist ")" | "*" .

forcedmethod = forcedname "(" forcedarglist ")" .
```

```
arglist = forcedarglist | "*" .

arglistopt = "," forcedname arglistopt | none .

forcedarglist = forcedname arglistopt | none .

inherit = "inherits" name | none .

forcedinherit = "inherits" forcedname | "implement" forcedname .
```

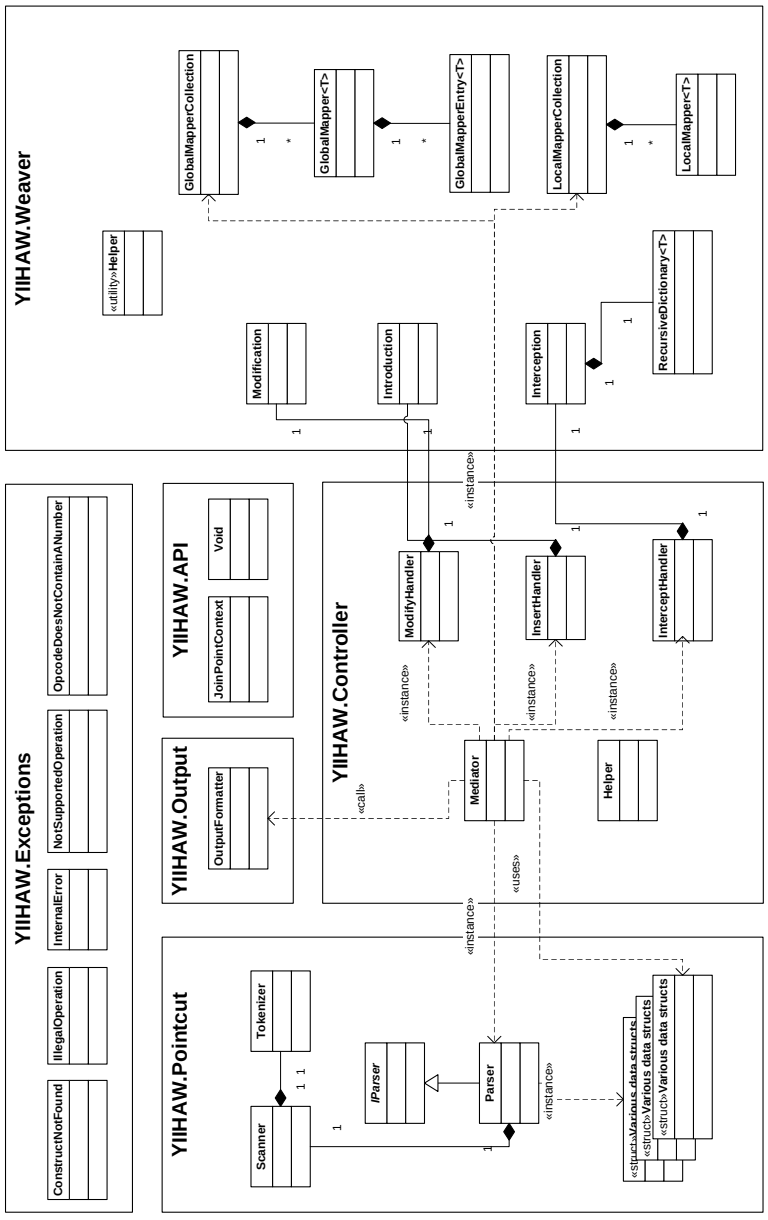
Appendix C

Short form notation for the pointcut language

Fully qualified name	Short form
System.Byte	byte
System.SByte	sbyte
System.Int32	int
System.UInt32	uint
System.Int16	short
System.UInt16	ushort
System.Int64	long
System.UInt64	ulong
System.Single	float
System.Double	double
System.Char	char
System.Boolean	bool
System.Object	object
System.String	string
System.Decimal	decimal

Appendix D

Class diagram



Appendix E

Source code for tests - Basecode

Test 1-3

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1
{
    public class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 4

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest4
{
    public class Tester
    {
        public static Random r = new Random();
```

```
        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted(r);
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted(Random r)
        {
            r.Next();
        }
    }
}
```

Test 4 static (for Aspect.NET)

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest4
{
    public class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                ToBeIntercepted(r);
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public static void ToBeIntercepted(Random r)
        {
            r.Next();
        }
    }
}
```

Test 5

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest5
{
```

```
    public class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                Tester.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public static void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 6

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest6
{
    public class Tester
    {
        public static Random r = new Random();

        public void GetNextInt()
        {
            r.Next();
        }
    }
}
```

Test 7

```
using System;
using System.Collections.Generic;
using System.Text;
using DotNetGuru.AspectDNG;
using DotNetGuru.AspectDNG.Joinpoints;


namespace WeaverTest7
{
    public abstract class SuperClass
    {
        public static Random r = new Random();
        public abstract int GetNextInt();
    }

    public class SubA : SuperClass
    {
```

```
        public override int GetNextInt()
        {
            return r.Next();
        }
    }

    public class Tester : SubA
    {
        static void Main(string[] args)
        {
            SuperClass t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.GetNextInt();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }
    }
}
```

Appendix F

Source code for tests - Coded by hand

Test 1-3

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1HW
{
    class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.NextDouble();
            r.Next();
        }
    }
}
```

Test 4

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest4HW
{
```

```
    public class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted(r);
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted(Random r)
        {
            r.NextDouble();
            r.Next();
        }
    }
}
```

Test 5

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest5HW
{
    public class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                Tester.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public static void ToBeIntercepted()
        {
            r.NextDouble();
            r.Next();
        }
    }
}
```

Test 6

```
using System;
```

```
using System.Collections.Generic;
using System.Text;

namespace WeaverTest6HW
{
    public class Tester
    {
        static void Main(string[] args)
        {
            WeaverTest6.Tester t = new WeaverTest6.Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.GetNextInt();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }
    }
}
```

Test 7

```
using System;
using System.Collections.Generic;
using System.Text;
using DotNetGuru.AspectDNG;
using DotNetGuru.AspectDNG.Joinpoints;


namespace WeaverTest7
{
    public abstract class SuperClass
    {
        public static Random r = new Random();
        public abstract int GetNextInt();
    }

    public class SubA : SuperClass
    {
        public override int GetNextInt()
        {
            return r.Next();
        }
    }

    public class SubB : SuperClass
    {
        public override int GetNextInt()
        {
            return r.Next();
        }
    }

    public class Tester : SubB
    {
        static void Main(string[] args)
        {
            SuperClass t = new Tester();
```

```
            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.GetNextInt();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }
    }
}
```

Appendix G

Source code for tests - AspectDNG

Test 1,5,6,7 - advice

```
using System;
using DotNetGuru.AspectDNG.Joinpoints;

// WeaverTest1 - around interception - pointcut specified via XML file
namespace AspectDNGWeaverTest1
{
    class Aspects
    {
        public static object Interceptor(JoinPoint jp)
        {
            WeaverTest1.Tester.r.NextDouble();
            return jp.Proceed(); // invoke original method
        }
    }
}

// WeaverTest2 - before interception - not possible in AspectDNG

// WeaverTest3 - after interception - not possible in AspectDNG

// WeaverTest4 - around interception with access to method arguments is not
    possible in AspectDNG

// WeaverTest5 - around interception of static method
namespace AspectDNGWeaverTest5
{
    class Aspects
    {
        public static object Interceptor(JoinPoint jp)
        {
            WeaverTest5.Tester.r.NextDouble();
            return jp.Proceed(); // invoke original method
        }
    }
}

// WeaverTest6 - method introduction
namespace AspectDNGWeaverTest6
{
    public class Methods
    {
        public int GetNextInt()
        {
```

```
            return WeaverTest6.Tester.r.Next();
        }
    }
}

// WeaverTest7 - typestructure modification - no implementation needed - specified
     via configuration file
```

Test 1 - configuration file

```
<AspectDngConfig warnings="$(path)/Warnings.log"
 weaving="$(path)/Weaving-log.xml" debug="true">
  <Variables>
    <Variable name="path" value="." />
    <Variable name="ns" value="" />
  </Variables>
  <TargetAssembly>$(path)/WeaverTest1.exe</TargetAssembly>
  <AspectsAssembly>$(path)/AspectDNGWeaverTest1.dll</AspectsAssembly>
  <WeavedAssembly>$(path)/Weaved.exe</WeavedAssembly>
  <PrivateLocations>
    <PrivatePath>$(path)</PrivatePath>
  </PrivateLocations>
  <Advice>
    <AroundBody targetXPath="//Method[match('* WeaverTest1.Tester::ToBeIntercepted(*)')]"
 aspectXPath="//Type[. = 'AspectDNGWeaverTest1.Aspects']/Method[@Name = 'Interceptor']" />
  </Advice>
</AspectDngConfig>
```

Test 5 - configuration file

```
<AspectDngConfig warnings="$(path)/Warnings.log"
 weaving="$(path)/Weaving-log.xml" debug="true">
  <Variables>
    <Variable name="path" value="." />
    <Variable name="ns" value="" />
  </Variables>
  <TargetAssembly>$(path)/WeaverTest5.exe</TargetAssembly>
  <AspectsAssembly>$(path)/AspectDNGWeaverTest1.dll</AspectsAssembly>
  <WeavedAssembly>$(path)/Weaved.exe</WeavedAssembly>
  <PrivateLocations>
    <PrivatePath>$(path)</PrivatePath>
  </PrivateLocations>
  <Advice>
    <AroundBody targetXPath="//Method[match('* WeaverTest5.Tester::ToBeIntercepted(*)')]"
 aspectXPath="//Type[. = 'AspectDNGWeaverTest5.Aspects']/Method[@Name = 'Interceptor']" />
  </Advice>
</AspectDngConfig>
```

Test 6 - configuration file

```
<AspectDngConfig warnings="$(path)/Warnings.log"
```

```
 weaving="$(path)/Weaving-log.xml" debug="true">
  <Variables>
    <Variable name="path" value="." />
    <Variable name="ns" value="" />
  </Variables>
  <TargetAssembly>$(path)/WeawerTest6TargetCode.dll</TargetAssembly>
  <AspectsAssembly>$(path)/AspectDNGWeaverTest1.dll</AspectsAssembly>
  <WeavedAssembly>$(path)/Weaved.exe</WeavedAssembly>
  <PrivateLocations>
    <PrivatePath>$(path)</PrivatePath>
  </PrivateLocations>
  <Advice>
    <Insert targetRegExp="WeaverTest6.Tester"
 aspectXPath="//Type[. = 'AspectDNGWeaverTest6.Methods']
/Method[@Name = 'GetNextInt']"/>
  </Advice>
</AspectDngConfig>
```

Test 7 - configuration file

```
<AspectDngConfig warnings="$(path)/Warnings.log"
 weaving="$(path)/Weaving-log.xml" debug="true">
  <Variables>
    <Variable name="path" value="." />
    <Variable name="ns" value="" />
  </Variables>
  <TargetAssembly>$(path)/WeaverTest7.exe</TargetAssembly>
  <AspectsAssembly>$(path)/AspectDNGWeaverTest1.dll</AspectsAssembly>
  <WeavedAssembly>$(path)/Weaved.exe</WeavedAssembly>
  <PrivateLocations>
    <PrivatePath>$(path)</PrivatePath>
  </PrivateLocations>
  <Advice>
    <SetBaseType targetRegExp="WeaverTest7.Tester"
 aspectXPath="//Type[. = 'AspectDNGWeaverTest7.SubB']"/>
  </Advice>
</AspectDngConfig>
```

Appendix H

Source code for tests - Aspect.NET

Test 2

```
using System;
using AspectDotNet;

public class Test2 : Aspect
{
    [AspectAction("%before %call ToBeIntercepted()")]
    public static void test2Aspect()
    {
        WeaverTest1.Tester.r.NextDouble();
    }
}
```

Test 3

```
using System;
using AspectDotNet;

public class Test3 : Aspect
{
    [AspectAction("%after %call ToBeIntercepted()")]
    public static void test3Aspect() {

        WeaverTest1.Tester.r.NextDouble();

    }
}
```

Test 4

```
using System;
using AspectDotNet;
using WeaverTest4;

public class Test4 : Aspect
{
    [AspectAction("%instead %call ToBeIntercepted(Random) && args(..)")]
    public static void test4Aspect(Random r) {

       WeaverTest4.Tester.ToBeIntercepted(r);
       Random r2 = new Random();
       r.NextDouble();
    }
```

```
}
```

Test 5

```
using System;
using AspectDotNet;
using WeaverTest5;

public class Test5 : Aspect
{
    [AspectAction("%instead %call ToBeIntercepted()")]
    public static void test5Aspect_() {
        WeaverTest5.Tester.ToBeIntercepted();
        Random r2 = new Random();
        r2.NextDouble();
    }
}
```

Appendix I

Source code for tests - NKalore

Test 1

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1
{
  public aspect AroundAspect
  {
    pointcut ToBeInterceptedPointCut void Tester.ToBeIntercepted();

    around void ToBeInterceptedPointCut()
    {
      Tester.r.NextDouble();
      proceed();
    }
  }

    class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 2

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1
{
  public aspect BeforeAspect
  {
    pointcut ToBeInterceptedPointCut void Tester.ToBeIntercepted();

    before ToBeInterceptedPointCut()
    {
            Tester.r.NextDouble();
    }
  }

    class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 3

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1
{
  public aspect AfterAspect
  {
    pointcut ToBeInterceptedPointCut void Tester.ToBeIntercepted();

    after ToBeInterceptedPointCut()
    {
            Tester.r.NextDouble();
    }
  }
```

```
    class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 4

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest4
{
  public aspect AroundAspect
  {
    pointcut ToBeInterceptedPointCut void Tester.ToBeIntercepted(Random r);

    around void ToBeInterceptedPointCut(Random r)
    {
        Tester.r.NextDouble();
      proceed(r);
    }
  }

    class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Tester t = new Tester();

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted(r);
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }
```

```
        public void ToBeIntercepted(Random r)
        {
            r.Next();
        }
    }
}
```

Test 5

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest5
{
  public aspect AroundAspect
  {
    pointcut ToBeInterceptedPointCut void Tester.ToBeIntercepted();

    around void ToBeInterceptedPointCut()
    {
            Tester.r.NextDouble();
      proceed();
    }
  }

    class Tester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                Tester.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public static void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Appendix J

Source code for tests - Rapier LOOM

Test 1

Aspect.cs

```
using System;
using Loom;
using System.Text;

namespace WeaverTest1
{
   public class Aspect1 : Loom.Aspect
    {
        [Loom.ConnectionPoint.Include("ToBeIntercepted")]
        [Loom.Call(Invoke.Instead)]
        public void aspect1()
        {
            Context.Invoke();
            Tester.r.Next();
        }
    }
}
```

ITester.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1
{
    public interface ITester
    {
        void ToBeIntercepted();
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Loom;

namespace WeaverTest1
{
```

```
    public class Tester : ITester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Aspect1 aspect = new Aspect1();
            ITester t = (ITester)Loom.Weaver.CreateInstance(typeof(Tester), null,
                aspect);

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 2

Aspect.cs

```
using System;
using Loom;
using System.Text;

namespace WeaverTest1
{
    public class Aspect2 : Loom.Aspect
    {
        [Loom.ConnectionPoint.Include("ToBeIntercepted")]
        [Loom.Call(Invoke.Before)]
        public void aspect2()
        {
            Tester.r.NextDouble();
        }
    }
}
```

ITester.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1
{
    public interface ITester
    {
        void ToBeIntercepted();
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Loom;

namespace WeaverTest1
{
    public class Tester : ITester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Aspect2 aspect = new Aspect2();
            ITester t = (ITester)Loom.Weaver.CreateInstance(typeof(Tester), null,
                aspect);

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();

            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 3

Aspect.cs

```
using System;
using Loom;
using System.Text;

namespace WeaverTest1
{
    public class Aspect3 : Loom.Aspect
    {
        [Loom.ConnectionPoint.Include("ToBeIntercepted")]
        [Loom.Call(Invoke.After)]
        public void aspect2()
        {
            Tester.r.NextDouble();
        }
    }
}
```

ITester.cs

```
using System;
```

```
using System.Collections.Generic;
using System.Text;

namespace WeaverTest1
{
    public interface ITester
    {
        void ToBeIntercepted();
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Loom;

namespace WeaverTest1
{
    public class Tester : ITester
    {
        public static Random r = new Random();

        static void Main(string[] args)
        {
            Aspect3 aspect = new Aspect3();
            ITester t = (ITester)Loom.Weaver.CreateInstance(typeof(Tester), null,
                aspect);


            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted()
        {
            r.Next();
        }
    }
}
```

Test 4

Aspect.cs

```
using System;
using Loom;
using System.Text;

namespace WeaverTest4
{
    public class Aspect4 : Loom.Aspect
    {
        [Loom.ConnectionPoint.Include("ToBeIntercepted")]
        [Loom.Call(Invoke.Instead)]
```

```
        public void aspect4(Random r)
        {
            Object[] ob = new Object[1];
            ob[0] = r;
            Context.Invoke(ob);
            r.NextDouble();
        }
    }
}
```

ITester.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest4
{
    public interface ITester
    {
        void ToBeIntercepted(Random r);
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Loom;

namespace WeaverTest4
{
    public class Tester : ITester
    {

        public static Random r = new Random();

        static void Main(string[] args)
        {
            Aspect aspect = new Aspect4();
            ITester t = (ITester)Loom.Weaver.CreateInstance(typeof(Tester), null,
                aspect);

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.ToBeIntercepted(r);
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }

        public void ToBeIntercepted(Random r)
        {
            r.Next();
        }
    }
}
```

Test 6

Aspect.cs

```
using System;
using Loom;
using System.Text;

namespace WeaverTest6
{
    [Loom.Introduces(typeof(IGetNextInt))]
    public class Aspect6 : Loom.Aspect, IGetNextInt
    {
        public void getNextInt()
        {
            Tester.r.Next();

        }
    }
}
```

IGetNextInt.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WeaverTest6
{
    public interface IGetNextInt
    {
        void getNextInt();
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Loom;

namespace WeaverTest6HW
{
    class Tester
    {
        static void Main(string[] args)
        {
            Aspect aspect = new WeaverTest6.Aspect6();
            WeaverTest6.IGetNextInt t = (WeaverTest6.IGetNextInt)Loom.Weaver.
                CreateInstance(typeof(WeaverTest6.Tester), null, aspect);

            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch
                ();
            watch.Start();
            for (int i = 0; i < 10000000; i++)
                t.getNextInt();
            watch.Stop();
            Console.WriteLine("time: " + watch.ElapsedMilliseconds + "
                milliseconds");
        }
    }
```

```
}
```

Appendix K

Source code for tests - YIIHAW

Test 1,4,5,6,7 - advice

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.RuntimeTests
{
    class Test1
    {
        public T Advice<T>()
        {
            WeaverTest1.Tester.r.NextDouble();
            return YIIHAW.API.JoinPointContext.Proceed<T>();
        }
    }

    class Test4
    {
        public T Advice<T>(Random r)
        {
            r.NextDouble();
            return YIIHAW.API.JoinPointContext.Proceed<T>();
        }
    }

    class Test5
    {
        public static T Advice<T>()
        {
            WeaverTest5.Tester.r.NextDouble();
            return YIIHAW.API.JoinPointContext.Proceed<T>();
        }
    }

    class Test6
    {
        public void GetNextInt()
        {
            WeaverTest6.Tester.r.Next();
        }
    }

    public class SubB : WeaverTest7.SuperClass
    {
```

```
        public override int GetNextInt()
        {
            return WeaverTest7.SuperClass.r.Next();
        }
    }
}
```

Test 1 - configuration file

```
around public instance void WeaverTest1.Tester:ToBeIntercepted()
do YIIHAW.RuntimeTests.Test1:Advice;
```

Test 4 - configuration file

```
around public instance void WeaverTest4.Tester:ToBeIntercepted(*)
do YIIHAW.RuntimeTests.Test4:Advice;
```

Test 5 - configuration file

```
around public static void WeaverTest5.Tester:ToBeIntercepted(*)
do YIIHAW.RuntimeTests.Test5:Advice;
```

Test 6 - configuration file

```
insert method public * void YIIHAW.RuntimeTests.Test6:GetNextInt()
into WeaverTest6.Tester;
```

Test 7 - configuration file

```
insert class YIIHAW.RuntimeTests.SubB into WeaverTest7;
modify WeaverTest7.Tester inherit YIIHAW.RuntimeTests.SubB;
```

Appendix L

Source code for collection tests - AspectDNG

Classes.cs

```
using System;
using System.Text;
using System.Reflection;

namespace AspectGenNonGeneric.Classes
{
    class LinkedListEnumerator : Collections.IEnumerator
    {
        Collections.LinkedList lst;
        Collections.LinkedList.Node curr;
        int stamp;
        bool valid;
        object item;

        public LinkedListEnumerator(Collections.LinkedList lst)
        {
            this.lst = lst;
            this.stamp = GetStampField(lst);

            Reset();
        }

        public object Current
        {
            get
            {
                if (valid)
                    return item;
                else
                    throw new InvalidOperationException();
            }
        }

        public bool MoveNext()
        {
            if (stamp != GetStampField(lst))
                throw new InvalidOperationException(); // List modified
            else if (curr != null)
            {
                item = curr.item;
```

```
            curr = curr.next;
            return valid = true;
        }
        else
            return valid = false;
    }

    public void Reset()
    {
        curr = lst.first;
        valid = false;
    }

    private int GetStampField(object o)
    {
        FieldInfo field = o.GetType().GetField("stamp",BindingFlags.NonPublic
            | BindingFlags.Instance); // reflectively get the "stamp" member
            of LinkedList
        if (field != null)
            return (int)field.GetValue(o);  // cast the "stamp" field to an
                int
        else
            throw new Exception("Could not retrieve the field \"stamp\"");
    }
}

class ArrayListEnumerator : Collections.IEnumerator
{
    Collections.ArrayList lst;
    bool valid;
    int stamp;
    object item;
    int curr;

    public ArrayListEnumerator(Collections.ArrayList lst)
    {
        this.lst = lst;
        stamp = GetStampField(lst); Reset();
    }

    public object Current
    {
        get
        {
            if (valid)
                return item;
            else
                throw new InvalidOperationException();
        }
    }

    public bool MoveNext()
    {
        if (stamp != GetStampField(lst))
            throw new InvalidOperationException();
        else if (curr < lst.size)
        {
            item = lst[curr];
            curr++;
            return valid = true;
        }
        else
```

```
                return valid = false;
            }

            public void Reset()
            {
                curr = 0;
                valid = false;
            }

            private int GetStampField(object o)
            {
                FieldInfo field = o.GetType().GetField("stamp", BindingFlags.NonPublic
                    | BindingFlags.Instance); // reflectively get the "stamp" member
                    of LinkedList
                if (field != null)
                    return (int)field.GetValue(o);  // cast the "stamp" field to an
                        int
                else
                    throw new Exception("Could not retrieve the field \"stamp\"");
            }
        }
    }
```

Fields.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AspectGenNonGeneric.Fields
{
    public class Fields
    {
        internal int stamp;
        public event EventHandler Changed;
    }
}
```

HelperInterfaces.cs

```
namespace AspectGenNonGeneric.Interfaces.Helper
{
    internal interface IOnChanged
    {
        void OnChanged(System.EventArgs e);
    }
}
```

Interceptors.cs

```
using DotNetGuru.AspectDNG.Joinpoints;
using System.Reflection;
using System;

namespace AspectGenNonGeneric.Interceptors
{
    class Interceptors
    {
        public static object AddCallToOnChangedAtEnd(JoinPoint jp)
        {
```

```
            object result = jp.Proceed();
            object o = jp.RealTarget;
            Interfaces.Helper.IOnChanged onchanged = (Interfaces.Helper.IOnChanged
                )o;
            onchanged.OnChanged(System.EventArgs.Empty);
            return result;
        }

        public static object UpdateStamp(JoinPoint jp)
        {
            object result = jp.Proceed();
            object o = jp.RealTarget;

            // get the "stamp"-field using reflection
            FieldInfo field = o.GetType().GetField("stamp", BindingFlags.NonPublic
                | BindingFlags.Instance); // reflectively get the "stamp" member
                of LinkedList
            if (field != null)
            {
                int stamp = (int)field.GetValue(o);  // cast the "stamp" field to
                    an int
                stamp++;
            }
            else
                throw new Exception("Could not retrieve the field \"stamp\"");

            return result;
        }
    }
}
```

Methods.cs

```
using System.Reflection;
using System;

namespace AspectGenNonGeneric.Methods
{
    class IOnChangedMethodsLinkedList : Interfaces.Helper.IOnChanged
    {
        public void OnChanged(System.EventArgs e)
        {
            object _this = this;
            FieldInfo field = ((Collections.LinkedList)_this).GetType().GetField("
                Changed", BindingFlags.Public | BindingFlags.Instance |
                BindingFlags.NonPublic); // reflectively get the "Changed"
                eventhandler
            if (field != null)
            {
                EventHandler handler = (EventHandler)field.GetValue(this);  //
                    cast the "Changed" field to a System.EventHandler
                if (handler != null)
                    handler(this, e);
            }
        }
    }

    class IOnChangedMethodsArrayList : Interfaces.Helper.IOnChanged
    {
        public void OnChanged(System.EventArgs e)
        {
```

```
            object _this = this;
            FieldInfo field = ((Collections.ArrayList)_this).GetType().GetField("
                Changed", BindingFlags.Public | BindingFlags.Instance |
                BindingFlags.NonPublic); // reflectively get the "Changed"
                eventhandler
            if (field != null)
            {
                EventHandler handler = (EventHandler)field.GetValue(this);  //
                    cast the "Changed" field to a System.EventHandler
                if (handler != null)
                    handler(this, e);
            }
        }
    }


    class IEnumerableMethodsLinkedList : Collections.LinkedList, Collections.
        IEnumerable
    {
        public Collections.IEnumerator GetEnumerator()
        {
            return new AspectGenNonGeneric.Classes.LinkedListEnumerator(this);
        }
    }

    class IEnumerableMethodsArrayList : Collections.ArrayList, Collections.
        IEnumerable
    {
        public Collections.IEnumerator GetEnumerator()
        {
            return new AspectGenNonGeneric.Classes.ArrayListEnumerator(this);
        }
    }

    class LinkedListEqualsMethods
    {
        public override bool Equals(object that)
        {
            object _this = this;
            if (that is Collections.IList && ((Collections.LinkedList)_this).size
                == ((Collections.IList)that).Count)
            {
                Collections.LinkedList.Node thisnode = ((Collections.LinkedList)
                    _this).first;
                Collections.IEnumerator thatenm = ((Collections.IEnumerable)that).
                    GetEnumerator();
                while (thisnode != null)
                {
                    if (!thatenm.MoveNext())
                        throw new Exception("Impossible: LinkedList<T>.Equals");
                    // assert MoveNext() was true;  // because of the above size
                        test

                    if (!thisnode.item.Equals(thatenm.Current))
                        return false;
                    thisnode = thisnode.next;
                }

                // assert !MoveNext(); // because of the size test

                return true;
            }
```

```
            else
                return false;
        }
    }
}
```

PublicInterfaces.cs

```
namespace Collections
{
    public interface IEnumerator
    {
        object Current { get; }
        bool MoveNext();
        void Reset();
    }

    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }
}
```

Enumeration - pointcut file

```
<AspectDngConfig warnings="$(path)/Warnings.log"
weaving="$(path)/Weaving-log.xml" debug="true">
    <Variables>
        <Variable name="path" value="."/>
        <Variable name="ns" value=""/>
    </Variables>
    <TargetAssembly>$(path)/dll/Basecode.dll</TargetAssembly>
    <AspectsAssembly>$(path)/AspectGenNonGeneric.dll</AspectsAssembly>
    <WeavedAssembly>$(path)/dll/Basecode.dll</WeavedAssembly>
    <PrivateLocations><PrivatePath>$(path)</PrivatePath></PrivateLocations>
    <Advice>
        <! -- make the ICollection interface inherit IEnumerable -->
        <ImplementInterface targetRegExp="Collections.ICollection"
        aspectXPath="//Type[. = 'Collections.IEnumerable']" />

        <! --  LinkedList: add the "stamp" member, add the LinkedListEnumerator class,
        add the GetEnumerator() method and override the Equals() methods -->
        <Insert targetRegExp="Collections.LinkedList"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Fields.Fields']
        /Field[@Name = 'stamp']"/>

        <Insert targetRegExp="Collections.*"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Classes.LinkedListEnumerator']"/>

        <Insert targetRegExp="Collections.LinkedList"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Methods.IEnumerableMethodsLinkedList']
        /Method[@Name = 'GetEnumerator']"/>

        <Insert targetRegExp="Collections.LinkedList"
```

```
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Methods.LinkedListEqualsMethods']
    /Method[@Name = 'ListEquals']" />

    <!-- ArrayList: add the "stamp" member, add the ArrayListEnumerator class,
    add the GetEnumerator() method and override the Equals() methods -->
    <Insert targetRegExp="Collections.ArrayList"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Fields.Fields']
    /Field[@Name = 'stamp']"/>

<Insert targetRegExp="Collections.ArrayList"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Classes.ArrayListEnumerator']"/>

    <Insert targetRegExp="Collections.ArrayList"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Methods.IEnumerableMethodsArrayList']
    /Method[@Name = 'GetEnumerator']"/>


<Insert targetRegExp="Collections.ArrayList"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Methods.ArrayListEqualsMethods']
    /Method[@Name = 'ListEquals']" />

    <!-- LinkedList: add interceptor that updates the "stamp" member -->
    <AroundBody targetXPath="//Method[match('* Collections.LinkedList
    ::AddFirst(*)')]"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
    /Method[@Name = 'UpdateStamp']"/>

    <AroundBody targetXPath="//Method[match('* Collections.LinkedList
    ::Add(*)')]"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
    /Method[@Name = 'UpdateStamp']"/>

    <AroundBody targetXPath="//Method[match('* Collections.LinkedList
    ::AddLast(*)')]"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
    /Method[@Name = 'UpdateStamp']"/>

    <AroundBody targetXPath="//Method[match('* Collections.LinkedList
    ::RemoveFirst(*)')]"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
    /Method[@Name = 'UpdateStamp']"/>

    <AroundBody targetXPath="//Method[match('* Collections.LinkedList
    ::RemoveAt(*)')]"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
    /Method[@Name = 'UpdateStamp']"/>

    <AroundBody targetXPath="//Method[match('* Collections.LinkedList
    ::RemoveLast(*)')]"
    aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
```

```
        /Method[@Name = 'UpdateStamp']"/>

        <AroundBody targetXPath="//Method[match('* Collections.LinkedList
        ::Remove(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'UpdateStamp']"/>

        <AroundBody targetXPath="//Method[match('* Collections.LinkedList
        ::set_Item(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'UpdateStamp']"/>

        <!-- ArrayList: add interceptor that updates the "stamp" member -->
        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::Add(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'UpdateStamp']"/>

        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::RemoveAt(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'UpdateStamp']"/>

        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::Remove(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'UpdateStamp']"/>

        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::set_Item(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'UpdateStamp']"/>
    </Advice>
</AspectDngConfig>
```

Event - pointcut file

```
<AspectDngConfig warnings="$(path)/Warnings.log"
weaving="$(path)/Weaving-log.xml" debug="true">
    <Variables>
        <Variable name="path" value="."/>
        <Variable name="ns" value=""/>
    </Variables>
    <TargetAssembly>$(path)/dll/Basecode.dll</TargetAssembly>
    <AspectsAssembly>$(path)/AspectGenNonGeneric.dll</AspectsAssembly>
    <WeavedAssembly>$(path)/dll/Basecode.dll</WeavedAssembly>
    <PrivateLocations><PrivatePath>$(path)</PrivatePath></PrivateLocations>
    <Advice>
        <!-- add eventhandling code for LinkedList -->
        <ImplementInterface targetRegExp="Collections.LinkedList"
```

```
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interfaces.Helper.IOnChanged']"/>

<Insert targetRegExp="Collections.LinkedList"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Fields.Fields']
/Field[@Name = 'Changed']" />

<Insert targetRegExp="Collections.LinkedList"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Methods.IOnChangedMethodsLinkedList']
/Method[@Name = 'OnChanged']"/>

<!-- add calls to the OnChanged() method -->
<AroundBody targetXPath="//Method[match('* Collections.LinkedList::AddFirst(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>

<AroundBody targetXPath="//Method[match('* Collections.LinkedList
::Add(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>

<AroundBody targetXPath="//Method[match('* Collections.LinkedList
::AddLast(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>

<AroundBody targetXPath="//Method[match('* Collections.LinkedList
::RemoveFirst(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>


<AroundBody targetXPath="//Method[match('* Collections.LinkedList
::RemoveAt(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>

<AroundBody targetXPath="//Method[match('* Collections.LinkedList
::RemoveLast(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>

<AroundBody targetXPath="//Method[match('* Collections.LinkedList
::Remove(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>

<AroundBody targetXPath="//Method[match('* Collections.LinkedList
::set_Item(*)')]"
aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
/Method[@Name = 'AddCallToOnChangedAtEnd']"/>
```

```
        <!-- add eventhandling code for ArrayList -->
        <ImplementInterface targetRegExp="Collections.ArrayList"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interfaces.Helper.IOnChanged']"/>

        <Insert targetRegExp="Collections.ArrayList"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Fields.Fields']
        /Field[@Name = 'Changed']" />

        <Insert targetRegExp="Collections.ArrayList"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Methods.IOnChangedMethodsArrayList']
        /Method[@Name = 'OnChanged']"/>

        <!-- add calls to the OnChanged() method -->
        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::Add(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'AddCallToOnChangedAtEnd']"/>

        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::RemoveAt(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'AddCallToOnChangedAtEnd']"/>

        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::Remove(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'AddCallToOnChangedAtEnd']"/>

        <AroundBody targetXPath="//Method[match('* Collections.ArrayList
        ::set_Item(*)')]"
        aspectXPath="//Type[. = 'AspectGenNonGeneric.Interceptors.Interceptors']
        /Method[@Name = 'AddCallToOnChangedAtEnd']"/>
    </Advice>
</AspectDngConfig>
```

Appendix M

Source code for collection tests - YIIHAW

Enumeration.cs

```
using System;
using System.Text;
using System.Reflection;

namespace YIIHAW.CollectionTests.Enumeration
{
    class LinkedListEnumerator : IEnumerator
    {
        Collections.LinkedList lst;
        Collections.LinkedList.Node curr;
        int stamp;
        int listStamp;
        bool valid;
        object item;

        /// <summary>
        /// Constructs a LinkedListEnumerator object
        /// </summary>
        /// <param name="lst">A reference to the LinkedList object for which this
            enumerator should operate</param>
        public LinkedListEnumerator(Collections.LinkedList lst, int stamp, ref int
             listStamp)
        {
            this.lst = lst;
            this.stamp = stamp;
            this.listStamp = listStamp;

            Reset();
        }


        /// <summary>
        /// Fetches the current item in the enumerator
        /// </summary>
        public object Current
        {
            get
            {
                if (valid)
                    return item;
```

```
            else
                throw new InvalidOperationException();
        }
    }

    /// <summary>
    /// Skips to the next item in the enumerator
    /// </summary>
    /// <returns>A boolean indicating if any items are left in the enumerator
        </returns>
    public bool MoveNext()
    {
        if (stamp != listStamp)
            throw new InvalidOperationException(); // List modified
        else if (curr != null)
        {
            item = curr.item;
            curr = curr.next;
            return valid = true;
        }
        else
            return valid = false;
    }

    /// <summary>
    /// Restarts the enumerator
    /// </summary>
    public void Reset()
    {
        curr = lst.first;
        valid = false;
    }
}

class ArrayListEnumerator : IEnumerator
{
    Collections.ArrayList lst;
    bool valid;
    int stamp;
    int listStamp;
    object item;
    int curr;

    /// <summary>
    /// Constructs an ArrayListEnumerator object
    /// </summary>
    /// <param name="lst">A reference to the LinkedList object for which this
        enumerator should operate</param>
    public ArrayListEnumerator(Collections.ArrayList lst, int stamp, ref int
        listStamp)
    {
        this.lst = lst;
        this.stamp = stamp;
        this.listStamp = listStamp;
    }

    /// <summary>
    /// Fetches the current item in the enumerator
    /// </summary>
    public object Current
    {
        get
```

```
            {
                if (valid)
                    return item;
                else
                    throw new InvalidOperationException();
            }
        }

        /// <summary>
        /// Skips to the next item in the enumerator
        /// </summary>
        /// <returns>A boolean indicating if any items are left in the enumerator
            </returns>
        public bool MoveNext()
        {
            if (stamp != listStamp)
                throw new InvalidOperationException();
            else if (curr < lst.size)
            {
                item = lst[curr];
                curr++;
                return valid = true;
            }
            else
                return valid = false;
        }

        /// <summary>
        /// Restarts the enumerator
        /// </summary>
        public void Reset()
        {
            curr = 0;
            valid = false;
        }
    }
}
```

EnumerationConstructs.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAW.API;

namespace YIIHAW.CollectionTests.Enumeration
{
    public class LinkedListEnumerationConstructs : Collections.LinkedList,
        Enumeration.IEnumerable
    {
        internal int stamp; // to be inserted into the target assembly

        /// <summary>
        /// Advice method that updates the stamp field that is being inserted into
             the target assembly
        /// </summary>
        /// <typeparam name="T">The return type of the target method</typeparam>
        /// <returns>The value of the original target method</returns>
        public T UpdateStamp<T>()
        {
```

```
            T result = JoinPointContext.Proceed<T>(); // invoke the original
                target method

            stamp++; // update the "stamp" field

            return result;
        }

        public IEnumerator GetEnumerator()
        {
            return new LinkedListEnumerator(this, stamp, ref stamp);
        }

        public override bool Equals(object that)
        {
            object _this = this;
            if (that is Collections.IList && ((Collections.LinkedList)_this).size
                == ((Collections.IList)that).Count)
            {
                Collections.LinkedList.Node thisnode = ((Collections.LinkedList)
                    _this).first;
                IEnumerator thatenum = ((IEnumerable)that).GetEnumerator();
                while (thisnode != null)
                {
                    if (!thatenum.MoveNext())
                        throw new Exception("Impossible: LinkedList.Equals");

                    if (!thisnode.item.Equals(thatenum.Current))
                        return false;
                    thisnode = thisnode.next;
                }

                return true;
            }
            else
                return false;
        }
    }

    public class ArrayListEnumerationConstructs : Collections.ArrayList,
        Enumeration.IEnumerable
    {
        internal int stamp; // to be inserted into the target assembly

        /// <summary>
        /// Advice method that updates the stamp field that is being inserted into
             the target assembly
        /// </summary>
        /// <typeparam name="T">The return type of the target method</typeparam>
        /// <returns>The value of the original target method</returns>
        public T UpdateStamp<T>()
        {
            T result = JoinPointContext.Proceed<T>(); // invoke the original
                target method

            stamp++; // update the "stamp" field

            return result;
        }

        public override bool Equals(object that)
        {
```

```
            object _this = this;
            if (that is Collections.IList && ((Collections.ArrayList)_this).size
                == ((Collections.IList)that).Count)
            {
                IEnumerator thatenm = ((IEnumerable)that).GetEnumerator();
                for (int i = 0; i < size; i++)
                {
                    if (!thatenm.MoveNext())
                        throw new Exception("Impossible: LinkedList.Equals");
                    // assert MoveNext() returned true;  /// because of the size
                        test
                    if (!elems[i].Equals(thatenm.Current))
                        return false;
                }
                // assert !MoveNext();  /// because of the size test
                return true;
            }
            else
                return false;
        }

        public IEnumerator GetEnumerator()
        {
            return new ArrayListEnumerator(this, stamp, ref stamp);
        }
    }
}
```

EnumerationInterfaces.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.CollectionTests.Enumeration
{
    public interface IEnumerator
    {
        object Current { get; }
        bool MoveNext();
        void Reset();
    }

    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }
}
```

Event.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAW.API;

namespace YIIHAW.CollectionTests.Event
{
    class EventConstructs
    {
```

```
        public event EventHandler changed; // to be inserted into the target
            assembly

        /// <summary>
        /// Advice method that invokes the changed event
        /// </summary>
        /// <typeparam name="T">The return type of the target method</typeparam>
        /// <returns>The value of the original target method</returns>
        public T AddCallToOnChangedAtEnd<T> ()
        {
            //T result = JoinPointContext.Proceed<T>(); // invoke the original
                target method

            OnChanged(System.EventArgs.Empty); // invoke the OnChanged() method

            //return result;
            return JoinPointContext.Proceed<T>();
        }

        /// <summary>
        /// Invoke the changed event
        /// </summary>
        /// <param name="e">Arguments passed to the changed event</param>
        public void OnChanged(System.EventArgs e)
        {
            if (changed != null)
                changed(this, e);
        }
    }
}
```

Enumeration - pointcut file

```
// enumeration - common
insert class YIIHAW.CollectionTests.Enumeration.IEnumerator into Collections;

insert class YIIHAW.CollectionTests.Enumeration.IEnumerable into Collections;

//enumeration - linkedlist
insert field internal * int YIIHAW.CollectionTests.Enumeration.LinkedList-
EnumerationConstructs:stamp into Collections.LinkedList;

insert class YIIHAW.CollectionTests.Enumeration.LinkedListEnumerator
into Collections.LinkedList;

insert method public instance IEnumerator YIIHAW.CollectionTests.Enumeration.-
LinkedListEnumerationConstructs:GetEnumerator() into Collections.LinkedList;

around public instance bool Collections.LinkedList:Equals(object)
do YIIHAW.CollectionTests.Enumeration.LinkedListEnumerationConstructs:Equals;

modify Collections.LinkedList implement YIIHAW.CollectionTests.Enumeration.IEnumerable;

around public * * Collections.LinkedList:Add*(*) do YIIHAW.CollectionTests-
.Enumeration.LinkedListEnumerationConstructs:UpdateStamp;
```

```
around public * * Collections.LinkedList:Remove*(*) do YIIHAW.CollectionTests-
.Enumeration.LinkedListEnumerationConstructs:UpdateStamp;

around public * * Collections.LinkedList:set_Item(*) do YIIHAW.CollectionTests-
.Enumeration.LinkedListEnumerationConstructs:UpdateStamp;


// enumeration - arraylist
insert field internal * int YIIHAW.CollectionTests.Enumeration.ArrayListEnumeration-
Constructs:stamp into Collections.ArrayList;

insert class YIIHAW.CollectionTests.Enumeration.ArrayListEnumerator
into Collections.ArrayList;

insert method public instance IEnumerator YIIHAW.CollectionTests.Enumeration.-
ArrayListEnumerationConstructs:GetEnumerator() into Collections.ArrayList;

around public instance bool Collections.ArrayList:Equals(object)
do YIIHAW.CollectionTests.Enumeration.ArrayListEnumerationConstructs:Equals;

modify Collections.ArrayList implement YIIHAW.CollectionTests.Enumeration.IEnumerable;

around public * * Collections.ArrayList:Add*(*)
do YIIHAW.CollectionTests.Enumeration.ArrayListEnumerationConstructs:UpdateStamp;

around public * * Collections.ArrayList:Remove*(*)
do YIIHAW.CollectionTests.Enumeration.ArrayListEnumerationConstructs:UpdateStamp;

around public * * Collections.ArrayList:set_Item(*)
do YIIHAW.CollectionTests.Enumeration.ArrayListEnumerationConstructs:UpdateStamp;
```

Event - pointcut file

```
insert event public * System.EventHandler YIIHAW.CollectionTests.Event.-
EventConstructs:changed into Collections.LinkedList;

insert event public * System.EventHandler YIIHAW.CollectionTests.Event.-
EventConstructs:changed into Collections.ArrayList;

insert method public instance void YIIHAW.CollectionTests.Event.-
EventConstructs:OnChanged(System.EventArgs) into Collections.LinkedList;

insert method public instance void YIIHAW.CollectionTests.Event.-
EventConstructs:OnChanged(System.EventArgs) into Collections.ArrayList;

around public * * Collections.*:Add(int,object) do YIIHAW.CollectionTests.-
Event.EventConstructs:AddCallToOnChangedAtEnd;

around public * * Collections.*:Remove(object) do YIIHAW.CollectionTests.-
Event.EventConstructs:AddCallToOnChangedAtEnd;
```

```
around public * * Collections.*:RemoveAt(int) do YIIHAW.CollectionTests.-
Event.EventConstructs:AddCallToOnChangedAtEnd;

around public * * Collections.*:set_Item(*) do YIIHAW.CollectionTests.-
Event.EventConstructs:AddCallToOnChangedAtEnd;
```

Appendix N

Source code for collection tests - Coded by hand

Collections.cs

```
// Generic typesafe collections in Generic C#
// This program requires .Net version 2.0.
// Peter Sestoft (sestoft@dina.kvl.dk) 2001-12-02, 2003-11-23, 2004-07-26

// Changed by Rasmus Johansen (johansen@itu.dk) 2006-05-02.
// The changed is made, to make the collection smaller, is it is to be used in a
    small project.

// NOTE: FOR SERIOUS WORK, USE THE C5 GENERIC COLLECTION LIBRARY!
//  SEE: http://www.itu.dk/research/c5/

// For this code, see documentation in file collections.txt

// To create a module for use from other files, compile with
//   csc /t:module GCollections.cs

using System;
using System.Diagnostics;      // For exceptions

namespace Collections
{

// INTERFACES ================================================

// Enumerators ---------------------------------------------------
#if ENUM
public interface IEnumerator {

  object Current { get; }
  bool MoveNext();
  void Reset();
}

// Enumerables ---------------------------------------------------

public interface IEnumerable {
  IEnumerator GetEnumerator();
}
#endif
// Collections ---------------------------------------------------
```

```
public interface ICollection
#if ENUM
    : IEnumerable
#endif
{
  int Count { get; }
}

// Comparing two things ---------------------------------------------

public interface IComparer {
  int Compare(object v1, object v2);
}

// Comparing to type T ----------------------------------------------

public interface IComparable {
  int CompareTo(object that);
}

// Lists, stacks and queues -----------------------------------------

public interface IList : ICollection {
  bool Add(object item);
  bool Add(int i, object item);
  object Remove();
  object RemoveAt(int i);
  object Remove(object item);
  bool Contains(object item);         // using Equals
  object this[int index] { get; set; }
#if EVENT
//*********** EventCode Start ************//
  // An event that clients can use to be notified whenever the
  // elements of the list change.
  event EventHandler Changed;
#endif
}



// IMPLEMENTATIONS ==============================================

// Doubly-linked lists ----------------------------------------------

// Add(T) at end, Remove() from front; behaves like a queue (FIFO)

public class LinkedList : IList {
  int size;      // Number of elements in the list
#if ENUM
    int stamp;       // To detect modification during enumeration
#endif
  Node first, last;    // Invariant: first==null iff last==null

  private class Node {
    public Node prev, next;
    public object item;

    public Node(object item) {
      this.item = item;
    }
```

```
    public Node(object item, Node prev, Node next) {
      this.item = item; this.prev = prev; this.next = next;
    }

  }
#if EVENT
  //*********** EventCode Start ***********//

  // An event that clients can use to be notified whenever the
  // elements of the list change.
  public event EventHandler Changed;
#else
    private static event EventHandler Changed = null;
#endif
  // Invoke the Changed event; called whenever list changes

    [Conditional("EVENT")]
    protected virtual void OnChanged(EventArgs e)
  {
      if (Changed != null)
          Changed(this, e);
  }


  //*********** EventCode End ***********//


  public LinkedList() {
    first = last = null;
    size = 0;
#if ENUM
      stamp = 0;
#endif
  }

  public int Count {
    get { return size; }
  }


  public object this[int index] {
    get { return get(index).item; }
    set {
        get(index).item = value;
        OnChanged(EventArgs.Empty);
    }
  }

  private Node get(int n) {
    if (n < 0 || n >= size)
      throw new IndexOutOfRangeException();
    else if (n < size/2) {              // Closer to front
      Node node = first;
      for (int i=0; i<n; i++)
        node = node.next;
      return node;
    } else {                            // Closer to end
      Node node = last;
      for (int i=size-1; i>n; i--)
        node = node.prev;
      return node;
    }
```

```
  }

  public bool Add(object item) {
    return AddLast(item);
  }

  public bool AddFirst(object item) {
    if (first == null) // and thus last == null
      first = last = new Node(item);
    else {
      Node tmp = new Node(item, null, first);
      first.prev = tmp;
      first = tmp;
    }

    size++;
#if ENUM
    stamp++;
#endif

    OnChanged(EventArgs.Empty);

    return true;
  }

  public bool Add(int i, object item) {
    if (i == 0)
      return AddFirst(item);
    else if (i == size)
      return AddLast(item);
    else {
      Node node = get(i);
      // assert node.prev != null;
      Node newnode = new Node(item, node.prev, node);
      node.prev.next = newnode;
      node.prev = newnode;

      size++;
#if ENUM
      stamp++;
#endif

      OnChanged(EventArgs.Empty);
      return true;
    }
  }

  public bool AddLast(object item) {
    if (last == null) // and thus first = null
      first = last = new Node(item);
    else {
      Node tmp = new Node(item, last, null);
      last.next = tmp;
      last = tmp;
    }

    size++;
#if ENUM
      stamp++;
#endif
```

```
    OnChanged(EventArgs.Empty);

    return true;
  }

  public object Remove() {
    return RemoveFirst();
  }



  public object RemoveFirst() {
    if (first == null) // and thus last == null
      throw new IndexOutOfRangeException();
    else {
      size--;
#if ENUM
      stamp++;
#endif

      object item = first.item;
      first = first.next;
      if (first == null)
        last = null;
      else
        first.prev = null;

    OnChanged(EventArgs.Empty);
      return item;
    }
  }

  public object RemoveAt(int i) {
    Node node = get(i);
    if (node.prev == null)  //node is first
      first = node.next;
    else
      node.prev.next = node.next;
    if (node.next == null) //node is last
      last = node.prev;
    else
      node.next.prev = node.prev;

    size--;
#if ENUM
      stamp++;
#endif
    OnChanged(EventArgs.Empty);

    return node.item;
  }

  public object RemoveLast() {
    if (last == null) // and thus first == null
      throw new IndexOutOfRangeException();
    else {
      size--;
#if ENUM
      stamp++;
#endif

      object item = last.item;
```

```
        last = last.prev;
        if (last == null)
          first = null;
        else
          last.next = null;

      OnChanged(EventArgs.Empty);

        return item;
      }
    }

    public object Remove(object item) {
      Node node = first;
      while (node != null) {
        if (item.Equals(node.item)) {
          if (node.prev == null)
            first = node.next;
          else
            node.prev.next = node.next;
          if (node.next == null)
            last = node.prev;
          else
            node.next.prev = node.prev;

          size--;
#if ENUM
        stamp++;
#endif

          OnChanged(EventArgs.Empty);

          return node.item;
        }

        node = node.next;
      }
      throw new ElementNotFoundException();
    }

    public bool Contains(object item) {
      Node node = first;
      while (node != null) {
        if (item.Equals(node.item))
          return true;
        node = node.next;
      }
      return false;
    }

    public override int GetHashCode() {
      int sum = 0;
      Node node = first;
      while (node != null) {
        sum = 31 * sum + node.item.GetHashCode();
        node = node.next;
      }
      return sum;
    }

#if ENUM
    public override bool Equals(object that) {
```

```
    if (that is IList && this.size == ((IList)that).Count) {
      Node thisnode = this.first;
      IEnumerator thatenm = ((IList)that).GetEnumerator();
      while (thisnode != null) {
  if (!thatenm.MoveNext())
    throw new Exception("Impossible: LinkedList.Equals");
        // assert MoveNext() was true;  // because of the above size test

        if (!thisnode.item.Equals(thatenm.Current))
          return false;
        thisnode = thisnode.next;
      }

      // assert !MoveNext(); // because of the size test

      return true;
    } else
      return false;
  }

  public IEnumerator GetEnumerator() {
    return new LinkedListEnumerator(this);
  }

  class LinkedListEnumerator : IEnumerator {
    LinkedList lst;
    Node curr;
    int stamp;
    bool valid;
    object item;

    public LinkedListEnumerator(LinkedList lst) {
      this.lst = lst; this.stamp = lst.stamp; Reset();
    }

    public object Current {
      get {
  if (valid)
    return item;
  else
    throw new InvalidOperationException();
      }
    }

    public bool MoveNext() {
      if (stamp != lst.stamp)
  throw new InvalidOperationException(); // List modified
      else if (curr != null)  {
        item = curr.item;
        curr = curr.next;
        return valid = true;
      } else
        return valid = false;
    }

    public void Reset() {
      curr = lst.first;
      valid = false;
    }
  }
#else
     public override bool Equals(object that) {
```

```
    if (that is IList && this.size == ((IList)that).Count) {
            IList thatlist = (IList)that;
            Node thisnode = this.first;
            int index = 0;
            while (thisnode != null)
            {
                if (!thisnode.item.Equals(thatlist[index]))
                    return false;
                thisnode = thisnode.next;
                index++;
            }
      return true;
    } else
      return false;
  }
#endif
}

// Array lists ---------------------------------------------------

// Add(T) at end, Remove() from end; behaves like a stack, LIFO

public class ArrayList : IList {

  int size;      // Number of elements in list
#if ENUM
  int stamp;       // To detect modification during enumeration
#endif
  object[] elems;
#if EVENT
  //*********** EventCode Start ************//

  // An event that clients can use to be notified whenever the
  // elements of the list change.
  public event EventHandler Changed;
#else
    private static event System.EventHandler Changed = null;
#endif

  // Invoke the Changed event; called whenever list changes
  [Conditional("EVENT")]
  protected virtual void OnChanged(EventArgs e)
  {
      if (Changed != null)
          Changed(this, e);
  }


  //*********** EventCode End ************//


  public ArrayList() {
      size = 0;
#if ENUM
      stamp = 0;
#endif
    elems = new object[10];  // Initial capacity
  }

  private void reallocate(int newsize) {
    object[] newelems = new object[newsize];
    for (int i=0; i<size; i++)
```

```
      newelems[i] = elems[i];
    elems = newelems;
  }

  public int Count {
    get { return size; }
  }

  public object this[int index] {
    get { return elems[index]; }
    set { elems[index] = value;

    OnChanged(EventArgs.Empty);

    }
  }

  public bool Add(object item) {
    return AddLast(item);
  }

  public bool AddLast(object item) { // Add at end
    return Add(size, item);
  }

  public bool Add(int i, object item) {       // Add at position i
    if (i<0 || i>size)
      throw new IndexOutOfRangeException();
    else {
      if (size == elems.Length)
        reallocate(2 * size);
      // assert elems.Length > size;
      for (int j=size; j>i; j--) //moving the elems above inserting index
        elems[j] = elems[j-1];
      elems[i] = item;
      size++;
#if ENUM
      stamp++;
#endif

    OnChanged(EventArgs.Empty);
      return true;
    }
  }

  public object Remove() {            // Remove last
    return RemoveAt(size-1);
  }

  public object RemoveAt(int i) {       // Remove at index i
    if (i<0 || i>=size)
      throw new IndexOutOfRangeException();
    else {
      object item = elems[i];
      for (int j=i+1; j<size; j++)
        elems[j-1] = elems[j];
      elems[--size] = default(object); // To prevent space leaks
#if ENUM
      stamp++;
#endif

    OnChanged(EventArgs.Empty);
```

```
      return item;
    }
  }

  public object Remove(object item) {      // Search
    for (int i=0; i<size; i++)
        if (item.Equals(elems[i]))
        {

    OnChanged(EventArgs.Empty);
            return RemoveAt(i);
        }
    throw new ElementNotFoundException();
  }

  public bool Contains(object item) {
    for (int i=0; i<size; i++)
      if (item.Equals(elems[i]))
        return true;
    return false;
  }

  public override int GetHashCode() {
    int sum = 0;
    for (int i=0; i<size; i++)
      sum = 31 * sum + elems[i].GetHashCode();
    return sum;
  }

#if ENUM
  public override bool Equals(object that) {
    if (that is IList && this.size == ((IList)that).Count) {
      IEnumerator thatenm = ((IList)that).GetEnumerator();
      for (int i=0; i<size; i++) {
  if (!thatenm.MoveNext())
    throw new Exception("Impossible: LinkedList<T>.Equals");
        // assert MoveNext() returned true;  /// because of the size test
        if (!elems[i].Equals(thatenm.Current))
          return false;
      }
      // assert !MoveNext();  /// because of the size test
      return true;
    } else
      return false;
  }

  public IEnumerator GetEnumerator() {
    return new ArrayListEnumerator(this);
  }

  class ArrayListEnumerator : IEnumerator {
    ArrayList lst;
    bool valid;
    int stamp;
    object item;
    int curr;

    public ArrayListEnumerator(ArrayList lst) {
      this.lst = lst; stamp = lst.stamp; Reset();
    }
```

```
    public object Current {
      get {
  if (valid)
    return item;
  else
    throw new InvalidOperationException();
      }
    }

    public bool MoveNext() {
      if (stamp != lst.stamp)
  throw new InvalidOperationException();
      else if (curr < lst.size)  {
        item = lst[curr];
        curr++;
        return valid = true;
      } else
        return valid = false;
    }

    public void Reset() {
      curr = 0;
      valid = false;
    }
  }
#else
    public override bool Equals(object that)
    {
        if (that is IList && this.size == ((IList)that).Count)
        {
            IList thatlist = (IList)that;
            for (int i = 0; i < size; i++)
            {
                if (!elems[i].Equals(thatlist[i]))
                    return false;
            }
            return true;
        }
        else
            return false;
    }
#endif
}

// Exceptions ------------------------------------------------

class ElementNotFoundException : Exception {
  public ElementNotFoundException() : base() { }
  public ElementNotFoundException(string s) : base(s) { }
}

} // End of namespace GCollections
```

Appendix O

Source code for collection tests - Basecode

Collections.cs

```
// NonGeneric typesafe collections in C# based on GCollection by Peter Sestoft (
    sestoft@dina.kvl.dk)
// This program requires .Net version 2.0.
// Rasmus Johansen (johansen@itu.dk) and Stephan Spangenberg (spangenberg@itu.dk)

// The program is only written for testing purposes, and should not be used.
// For a more fullblown collection library, use the C5 Generic Collection Library
// See: http://www.itu.dk/research/c5/

using System;      // For exceptions

namespace Collections
{

    // INTERFACES ============================================================

    // Collections ---------------------------------------------------------

    public interface ICollection
    {
        int Count { get; }
    }

    // Comparing two things ------------------------------------------------

    public interface IComparer
    {
        int Compare(object v1, object v2);
    }

    // Comparing to type T -------------------------------------------------

    public interface IComparable
    {
        int CompareTo(object that);
    }

    // Lists, stacks and queues --------------------------------------------

    public interface IList : ICollection
```

```
{
    bool Add(object item);
    bool Add(int i, object item);
    object Remove();
    object RemoveAt(int i);
    object Remove(object item);
    bool Contains(object item);         // using Equals
    object this[int index] { get; set; }
}


// IMPLEMENTATIONS ============================================

// Doubly-linked lists ------------------------------------------

// Add(T) at end, Remove() from front; behaves like a queue (FIFO)

public class LinkedList : IList
{
    internal int size;       // Number of elements in the list
    public Node first, last;    // Invariant: first==null iff last==null

    public class Node
    {
        public Node prev, next;
        public object item;

        public Node(object item)
        {
            this.item = item;
        }

        public Node(object item, Node prev, Node next)
        {
            this.item = item; this.prev = prev; this.next = next;
        }

    }

    public LinkedList()
    {
        first = last = null;
        size = 0;
    }

    public int Count
    {
        get { return size; }
    }


    public object this[int index]
    {
        get { return get(index).item; }
        set { get(index).item = value; }
    }

    private Node get(int n)
    {
        if (n < 0 || n >= size)
            throw new IndexOutOfRangeException();
        else if (n < size / 2)
```

```
        {               // Closer to front
            Node node = first;
            for (int i = 0; i < n; i++)
                node = node.next;
            return node;
        }
        else
        {                               // Closer to end
            Node node = last;
            for (int i = size - 1; i > n; i--)
                node = node.prev;
            return node;
        }

    }

    public bool Add(object item)
    {
        return AddLast(item);
    }

    public bool AddFirst(object item)
    {
        if (first == null) // and thus last == null
            first = last = new Node(item);
        else
        {
            Node tmp = new Node(item, null, first);
            first.prev = tmp;
            first = tmp;
        }

        size++;
        return true;
    }

    public bool Add(int i, object item)
    {
        if (i == 0)
            return AddFirst(item);
        else if (i == size)
            return AddLast(item);
        else
        {
            Node node = get(i);
            // assert node.prev != null;
            Node newnode = new Node(item, node.prev, node);
            node.prev.next = newnode;
            node.prev = newnode;

            size++;

            return true;
        }
    }

    public bool AddLast(object item)
    {
        if (last == null) // and thus first = null
            first = last = new Node(item);
        else
        {
```

```
            Node tmp = new Node(item, last, null);
            last.next = tmp;
            last = tmp;
        }

        size++;

        return true;
    }

    public object Remove()
    {
        return RemoveFirst();
    }



    public object RemoveFirst()
    {
        if (first == null) // and thus last == null
            throw new IndexOutOfRangeException();
        else
        {
            size--;
            object item = first.item;
            first = first.next;
            if (first == null)
                last = null;
            else
                first.prev = null;
            return item;
        }
    }

    public object RemoveAt(int i)
    {
        Node node = get(i);
        if (node.prev == null)  //node is first
            first = node.next;
        else
            node.prev.next = node.next;
        if (node.next == null) //node is last
            last = node.prev;
        else
            node.next.prev = node.prev;

        size--;

        return node.item;
    }

    public object RemoveLast()
    {
        if (last == null) // and thus first == null
            throw new IndexOutOfRangeException();
        else
        {
            size--;
            object item = last.item;
            last = last.prev;
            if (last == null)
                first = null;
```

```
            else
                last.next = null;
            return item;
        }
    }

    public object Remove(object item)
    {
        Node node = first;
        while (node != null)
        {
            if (item.Equals(node.item))
            {
                if (node.prev == null)
                    first = node.next;
                else
                    node.prev.next = node.next;
                if (node.next == null)
                    last = node.prev;
                else
                    node.next.prev = node.prev;

                size--;

                return node.item;
            }

            node = node.next;
        }
        throw new ElementNotFoundException();
    }

    public bool Contains(object item)
    {
        Node node = first;
        while (node != null)
        {
            if (item.Equals(node.item))
                return true;
            node = node.next;
        }
        return false;
    }

    public override int GetHashCode()
    {
        int sum = 0;
        Node node = first;
        while (node != null)
        {
            sum = 31 * sum + node.item.GetHashCode();
            node = node.next;
        }
        return sum;
    }

    public override bool Equals(object that)
    {
        if (that is IList && this.size == ((IList)that).Count)
        {
            IList thatlist = (IList)that;
            Node thisnode = this.first;
```

```
            int index = 0;
            while (thisnode != null)
            {
                if (!thisnode.item.Equals(thatlist[index]))
                    return false;
                thisnode = thisnode.next;
                index++;
            }
            return true;
        }
        else
            return false;
    }
}

// Array lists ---------------------------------------------------

// Add(T) at end, Remove() from end; behaves like a stack, LIFO

public class ArrayList : IList
{

    public int size;       // Number of elements in list
    internal object[] elems;

    public ArrayList()
    {
        size = 0;
        elems = new object[10];  // Initial capacity
    }

    private void reallocate(int newsize)
    {
        object[] newelems = new object[newsize];
        for (int i = 0; i < size; i++)
            newelems[i] = elems[i];
        elems = newelems;
    }

    public int Count
    {
        get { return size; }
    }

    public object this[int index]
    {
        get { return elems[index]; }
        set { elems[index] = value; }
    }

    public bool Add(object item)
    {
        return AddLast(item);
    }

    public bool AddLast(object item)
    { // Add at end
        return Add(size, item);
    }

    public bool Add(int i, object item)
    {       // Add at position i
```

```
        if (i < 0 || i > size)
            throw new IndexOutOfRangeException();
        else
        {
            if (size == elems.Length)
                reallocate(2 * size);
            // assert elems.Length > size;
            for (int j = size; j > i; j--) //moving the elems above inserting
                index
                elems[j] = elems[j - 1];
            elems[i] = item;
            size++;
            return true;
        }
    }

    public object Remove()
    {           // Remove last
        return RemoveAt(size - 1);
    }

    public object RemoveAt(int i)
    {       // Remove at index i
        if (i < 0 || i >= size)
            throw new IndexOutOfRangeException();
        else
        {
            object item = elems[i];
            for (int j = i + 1; j < size; j++)
                elems[j - 1] = elems[j];
            elems[--size] = default(object); // To prevent space leaks
            return item;
        }
    }

    public object Remove(object item)
    {      // Search
        for (int i = 0; i < size; i++)
            if (item.Equals(elems[i]))
                return RemoveAt(i);
        throw new ElementNotFoundException();
    }

    public bool Contains(object item)
    {
        for (int i = 0; i < size; i++)
            if (item.Equals(elems[i]))
                return true;
        return false;
    }

    public override int GetHashCode()
    {
        int sum = 0;
        for (int i = 0; i < size; i++)
            sum = 31 * sum + elems[i].GetHashCode();
        return sum;
    }

    public override bool Equals(object that)
    {
        if (that is IList && this.size == ((IList)that).Count)
```

```
            {
                IList thatlist = (IList)that;
                for (int i = 0; i < size; i++)
                {
                    if (!elems[i].Equals(thatlist[i]))
                        return false;
                }
                return true;
            }
            else
                return false;
        }

    }

    // Exceptions ---------------------------------------------------

    class ElementNotFoundException : Exception
    {
        public ElementNotFoundException() : base() { }
        public ElementNotFoundException(string s) : base(s) { }
    }

} // End of namespace Collections
```

Appendix P

Source code for collection tests - Test program

CollectionTester

```
using System;
using System.Text;
using Collections;
using System.Reflection;

namespace OurCollectionTesterApp
{
    class Program
    {
        static void Main(string[] args)
        {
#if EVENT
            IList list1 = new LinkedList();
            IList list2 = new ArrayList();

            DateTime start = DateTime.Now;
            for (int i = 0; i < 10000; i++)
            {
                list1.Add(0, "string"+i);
                list1.Add(1, "string" + i + i);
                list2.Add(0, "string" + i);
            }
            for (int i = 0; i < 5000; i++)
            {
                list1.Remove();
                list2.Remove();
            }
            for (int i = 0; i < 2500; i++)
            {
                list1.RemoveAt(0);
                list2.RemoveAt(0);
            }
            string ssss = "ssss";
            string ss = "ss";
            for (int i = 0; i < 10000; i++)
            {
                list1.Add(ssss);
                list1.Add(ss);
                list2.Add(ssss);
                list2.Add(ss);
```

```
            }
            for (int i = 0; i < 2500; i++)
            {
                list1.Remove(ssss);
                list2.Remove(ssss);
            }

            DateTime end = DateTime.Now;
            TimeSpan dif = end.Subtract(start);
            Console.WriteLine("time: " + dif.TotalMilliseconds + " milliseconds");
#endif

#if ENUM
            LinkedList list1_enum = new LinkedList();
            ArrayList list2_enum = new ArrayList();

            for (int j = 0; j < 10000; j++)
            {
                list1_enum.Add("ggg" + j);
                list2_enum.Add("ggg" + j);
            }

            DateTime start_enum = DateTime.Now;

            for (int k = 0; k < 250; k++)
            {
                list1_enum.Equals(list2_enum);
                list2_enum.ListEquals(list1_enum);
            }

            DateTime end_enum = DateTime.Now;
            TimeSpan dif_enum = end_enum.Subtract(start_enum);
            Console.WriteLine("time: " + dif_enum.TotalMilliseconds + "
                milliseconds");

#endif

        }

    }
}
```

Appendix Q

Partial functional testing overview

Return semantic \ Return type	Primitive	String	External	Target	Void
Proceed – defined type	12	3	4	15	5
Proceed – generic type	17	17	17	17	11
Defined type without proceed	1	2	6	8	9
Default with proceed	18	18	18	18	18
Default without proceed	19	19	19	19	19

Figure Q.1: The tests of interception.

The interception pointcut syntax:

```
around <access> <invocation kind> <return type> <type>:<method(arguments)>
[inherits <type>] do <advice type>:<advice method>;
```

Access specification	Public	Private	Internal	Protected	*
Access specification	1	7	8	9	4
Invocation kind	Static	Instance	*		
Invocation kind	9	11	4		
Return type	Fully defined	Defined as primitive			*
Return type	3	1			2
Type name specification	*	name.name	*.name	Name.*	
Type name specification	6	3	1	7	
Method name specification	*	Name	*name	Name*	
	6	1	7	4	
Argument definition	Defined	Defined as primitive			*
Argument definition	6	9			4
Inherit defined	15				
Matching advice methods	One	More than one			
Matching advice methods	1	4			
Matching targets	One	More than one			
Matching targets	1	4			
Target which doesn't match		On returntype	On argument types		
Target which doesn't match		5	9		

Figure Q.2: The tests of the interception pointcut syntax.

The numbers in the tables indicates the test number.

Has reference to \ introducing construct	Method	Field	Property	Class	Event
Mscorlib	12	12	12	12	12
Target assembly	13	13	13	13	13
Aspect assembly	10	15	12	10	N/A
External assembly	16	16	16	16	16

Figure Q.3: The tests of introduction.

The introduction pointcut syntax:

```
insert <construct> <access> <invocation kind> [return type]
<aspect type>:<aspect name[(arguments)]> into <type>;
```

	Method	Field	Property	Class	Event
Access specifier = public	10	15	12		12
Access specifier = private	20	13	20		13
Access specifier = protected	16	16	16	N/A	16
Access specifier = internal	13	12	13		20
Access specifier = *	10	20	20		20
Invocation kind = static	16	13	16		13
Invocation kind = instance	12	15	13	N/A	12
Invocation kind = *	10	12	12		20
Returntype = void	11	N/A	N/A		N/A
Returntype = mscorlib	10	12	12		12
Returntype = target	20	13	13	N/A	13
Returntype = aspect	20	20	20		N/A
Returntype = external	16	16	16		16
Target = *	16	16	16	16	16
Target = *.xxx	13	13	13	20	13
Target = xxx.*	20	15	20	21	20
Target = xxx.yyy	10	12	12	12	12

Figure Q.4: The tests of the introduction pointcut syntax.

Implement interface	22
Change basetype	21

Figure Q.5: The tests of modification.

The modification pointcut syntax:

```
modify <type> <action> <aspect type>;
```

Action = inherit	21
Action = implement	22

Figure Q.6: The tests of the modification pointcut syntax.

The numbers in the tables indicates the test number.

Proceed<T>	1
GetTarget<T>	20
AccessSpecifier	23
Arguments	23
DeclaringType	23
IsStatic	23
Name	23
ReturnType	23

Figure Q.7: The tests of the *JoinPointContext*.

Use of newarr	20
Use of boxing	1
Use of unboxing	12
Use of new obj	5
Use of init obj	19
Use of call and virtcall	4
Accessing local variable	10
Accessing a field	13
Accessing an argument	6
Use of ldtoken with reference to class, method, field	14
Use of switch	23

Figure Q.8: The tests of special opcodes.

The numbers in the tables indicates the test number.

Appendix R

Source code for partial functional testing

Tester.cs - framework's main program file

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using System.IO;
using System.Diagnostics;

namespace YIIHAWTester
{
    /// <summary>
    /// The main class of the tester framework. Used to start the testing.
    /// </summary>
    public class Tester
    {
        public static ErrorLogger errorLogger;

        /// <summary>
        /// Starts the testing
        /// </summary>
        /// <param name="args">The arguments needed for the testing (A path and
            name of an assembly containing test methods).</param>
        static void Main(string[] args)
        {
            int numberOfTestMethods = 0;
            Assembly assembly;
            errorLogger = new ErrorLogger();

            if (args.Length > 0)
                assembly = Assembly.LoadFrom(args[0]);
            else //TODO: Remove - Just for easy debugging.
                assembly = Assembly.LoadFrom("../../../YIIHAWTests/bin/Debug/
                    YIIHAWTests.dll");

            foreach (Type type in assembly.GetTypes())
            {
                object[] attributes = type.GetCustomAttributes(typeof(
                    TestableClassAttribute), false);
                if (attributes.Length == 0)
                    continue;
```

```
            // If it does have the TestableClassAttribute,
            // run the tests in it
            numberOfTestMethods += RunTests(type);
        }
        errorLogger.Print("Number of test methods runned = "+
            numberOfTestMethods);
        Console.ReadLine();
    }

    /// <summary>
    /// Runs the test methods in a given type.
    /// </summary>
    /// <param name="type">The type which has the test methods to run.</param>
    /// <returns>The number of test methods runned in the given type.</returns
        >
    private static int RunTests(Type type)
    {
        object iTestAbleObject = Activator.CreateInstance(type);
        int numberOfTestMethods = 0;
        foreach (MethodInfo methodInfo in type.GetMethods(BindingFlags.
            Instance | BindingFlags.NonPublic | BindingFlags.Public))
        {
            //Check if this method is a test method by looking for the
                testableMethod attribute.
            object[] attributes = methodInfo.GetCustomAttributes(typeof(
                TestableMethodAttribute), false);
            if (attributes.Length == 0)
                continue;

            numberOfTestMethods++;
            if (!Directory.Exists("./outputFiles"))
                Directory.CreateDirectory("./outputFiles");


            FileInfo inputfile = new FileInfo((attributes[0] as
                TestableMethodAttribute).InputFile);
            string outputfileString = "./outputFiles/" + methodInfo.Name + "/"
                + methodInfo.Name + "Out" + inputfile.Extension;
            FileInfo outputfile = new FileInfo(outputfileString);
            if (!Directory.Exists("./outputFiles/" + methodInfo.Name))
                Directory.CreateDirectory("./outputFiles/" + methodInfo.Name);

            if (!inputfile.Exists)
            {
                Console.WriteLine("The specified inputfile: '{0}' for test
                    method: '{1}' does not exist", inputfile.Name, methodInfo.
                    Name);
                errorLogger.AddError("The specified inputfile: '" + inputfile.
                    FullName + "' for test method: '" + methodInfo.Name + "'
                    in class: '" + type.FullName + "' does not exist");
                continue;
            }

            Console.WriteLine("
                ************************************************************")
                ;
            Console.WriteLine("Invoking testmethod : '{0}'\ninputfile = '{1}'
                \noutputfile = '{2}'\n", methodInfo.Name, inputfile,
                outputfile);
            outputfile = null;
            try
            {
```

```
                type.InvokeMember(methodInfo.Name, BindingFlags.Instance |
                    BindingFlags.Static | BindingFlags.Public | BindingFlags.
                    NonPublic | BindingFlags.InvokeMethod, null,
                    iTestAbleObject, null);
            }
            catch (Exception e)
            {
                errorLogger.AddError(e.ToString());
                Break();
            }

            // If the test has an outputfile, do verification of the output
                file.
            if ((attributes[0] as TestableMethodAttribute).HasOutputFile)
            {
                outputfile = new FileInfo(outputfileString);
                if (!outputfile.Exists)
                {
                    Console.WriteLine("The specified outputfile: '{0}' for
                        test method: '{1}' does not exist", outputfile.Name,
                        methodInfo.Name);
                    errorLogger.AddError("The specified outputfile: '" +
                        outputfile.FullName + "' for test method: '" +
                        methodInfo.Name + "' in class: '" + type.FullName + "'
                         does not exist");
                    continue;
                }

                if (!VerifyAssembly(inputfile, outputfile, type.FullName +
                    methodInfo.Name))
                    continue;
            }
            inputfile = null;
            outputfile = null;
        }

        return numberOfTestMethods;
    }

    /// <summary>
    /// PeVerifies the given outputfile, and if it fails also verify the given
         inputfile.
    /// </summary>
    /// <param name="inputfile">The inputfile.</param>
    /// <param name="outputfile">The output file.</param>
    /// <param name="methodName">The name of the method where the given files
        are tested.</param>
    /// <returns>A boolean indicating if the verification was successful.</
        returns>
    private static bool VerifyAssembly(FileInfo inputfile, FileInfo outputfile
        , string methodName)
    {

        string outfileOutput = DoPeverify(outputfile);

        string outSuccesString = "All Classes and Methods in " + outputfile.
            FullName + " Verified.";
        if (!outfileOutput.Equals(outSuccesString)) //PEverify was not
            successful
        {
            Console.WriteLine("PeVerify on outputfile: '{0}' was not
                successful, the following error was given:\n{1}", outputfile.
```

```
            Name, outfileOutput);
        //Check if inputfile could be verified.
        string infileOutput = DoPeverify(inputfile);
        string inSuccesString = "All Classes and Methods in " + inputfile.
            FullName + " Verified.";
        if (!infileOutput.Equals(inSuccesString)) //PEverify on inputfile
            was not successful
        {
            Console.WriteLine("PeVerify on inputfile: '{0}' was not
                successful, the following error was given:\n{1}",
                inputfile.Name, infileOutput);
            errorLogger.AddError(
                "The verification of outputfile: '" + outputfile.FullName
                    +
                "' was not successful. Neither was the verification of the
                     inputfile:'" +
                inputfile.FullName + "'. This happend after test method: '
                    " + methodName + "'.\n The following messages was
                    returned from the verification:\n" +
                "Inputfile verfication:\n" + infileOutput + "\n" +
                "Outputfile verfication:\n" + outfileOutput);
            return false;
        }
        else
        {
            errorLogger.AddError(
                "The verification of outputfile: '" + outputfile.FullName
                    + "' was not successful in test method: '" +
                    methodName + "'." +
                " The following message was returned from the verification
                    :\n" + outfileOutput);
            return false;
        }
    }
    return true;
}

/// <summary>
/// Do Peverify on a given file.
/// </summary>
/// <param name="file">The file to do the verifications on.</param>
/// <returns>A string with the console output from the verification.</
    returns>
private static string DoPeverify(FileInfo file)
{
    ProcessStartInfo startInfo = new ProcessStartInfo("peverify");
    startInfo.Arguments = "\"" + file.FullName + "\" /nologo";
    startInfo.UseShellExecute = false;
    startInfo.RedirectStandardOutput = true;


    using (Process peverify = Process.Start(startInfo))
    {
        StreamReader outputReader = peverify.StandardOutput;
        peverify.Start();
        string output = outputReader.ReadLine();

        return output;
    }
}

/// <summary>
```

```
        /// Method used to break the testing, if there is a serious error.
        /// </summary>
        internal static void Break()
        {
            errorLogger.Print("The tester was stopped in the middle of testing");
            Console.ReadLine();
            Environment.Exit(0);
        }
    }
}
```

API.cs - framework api

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Diagnostics;
using System.Reflection;

namespace YIIHAWTester
{
    /// <summary>
    /// This is the API class for the test framework.
    /// </summary>
    public class API
    {

        /// <summary>
        /// Creates a pointcut file with the given name.
        /// </summary>
        /// <param name="pointcut">The pointcut statement to be saved in the
            pointcut file.</param>
        /// <param name="filename">The name given to the pointcut file.</param>
        public static void CreatePointcutFile(string pointcut, string filename)
        {
            FileInfo newFile = new FileInfo(filename);
            newFile.Delete();

            FileStream fileStream = newFile.OpenWrite();
            TextWriter writer = new StreamWriter(fileStream);
            writer.WriteLine(pointcut);
            writer.Close();
            fileStream.Close();
        }

        /// <summary>
        /// Weave method, that checks if the output of the weaving is as expected.
        /// </summary>
        /// <param name="target">Target of the weaving.</param>
        /// <param name="aspect">Aspect of the weaving.</param>
        /// <param name="pointcutFile">The pointcut file to weave after.</param>
        /// <param name="expectedOutput">The expected output to check against.</
            param>
        public static void WeaveWithExpectedOutput(string target, string aspect,
            string pointcutFile, string expectedOutput)
        {
            StringBuilder output = Weave(target, aspect, pointcutFile, true);
            string outputFile = "./outputFiles/" + getCallerIDName() + "/" +
                getCallerIDName() + "Out." + target.Substring(target.Length - 3);
            checkWeaverExpectedOutput(output.ToString(), target, aspect,
```

```
            pointcutFile, outputFile, expectedOutput);
        output = null;

    }

    /// <summary>
    /// Weave method. Expects the weaving to be successful.
    /// </summary>
    /// <param name="target">Target of the weaving.</param>
    /// <param name="aspect">Aspect of the weaving.</param>
    /// <param name="pointcutFile">The pointcut file to weave after.</param>
    public static void Weave(string target, string aspect, string pointcutFile
        )
    {
        Weave(target, aspect, pointcutFile, false);
    }

    /// <summary>
    /// The actual method that calls the weaver.
    /// </summary>
    /// <param name="target">Target of the weaving.</param>
    /// <param name="aspect">Aspect of the weaving.</param>
    /// <param name="pointcutFile">The pointcut file to weave after.</param>
    /// <param name="returnWeaverOutput">Indicate whether the method, should
        return the console output from the weaver or not.</param>
    /// <returns>If the "returnWeaverOutput" parameter is true, a
        StringBuilder with the console output of the weaver will be returned.
        Else null.</returns>
    private static StringBuilder Weave(string target, string aspect, string
        pointcutFile, bool returnWeaverOutput)
    {
        //Build up the arguments to the main method of the weaver.
        string[] args = new string[5];
        args[0] = pointcutFile;
        args[1] = target;
        args[2] = aspect;
        string outputFile = "./outputFiles/" + getCallerIDName() + "/" +
            getCallerIDName() + "Out." + target.Substring(target.Length - 3);
        args[3] = outputFile;
        args[4] = "-v";

        //Change the console.out so that it can be catched.
        TextWriter oldOut = Console.Out;
        StringBuilder output = new StringBuilder();
        TextWriter thisWriter = new StringWriter(output);

        Console.SetOut(thisWriter); //the actual change
        try
        {
            YIIHAW.Controller.Mediator.Main(args);
        }
        catch (Exception e)
        {
            Tester.errorLogger.AddError(e.ToString());
        }
        Console.SetOut(oldOut); //change it back after the weaving.

        if (returnWeaverOutput)
            return output;

        if (!checkWeaverOutput(output.ToString(), target, aspect, pointcutFile
            , outputFile))
```

```
            Tester.Break();

        Console.WriteLine("***************** Start on output from weaver
            *****************");
        Console.WriteLine(output);
        Console.WriteLine("***************** End of output from weaver
            *****************\n");
        return null;
    }

    /// <summary>
    /// Checks if the weaving was successful or ended with a fatal error.
    /// </summary>
    /// <param name="output">The console output from the weaving.</param>
    /// <param name="target">Target of the weaving.</param>
    /// <param name="aspect">Aspect of the weaving.</param>
    /// <param name="pointcutFile">The pointcut file to weave after.</param>
    /// <param name="outputFile">The name of the file which might have been
        outputted by the weaving.</param>
    /// <returns>A boolean indicating if the weaving ended with success or not
        .</returns>
    private static bool checkWeaverOutput(string output, string target, string
         aspect, string pointcutFile, string outputFile)
    {
        if (output.StartsWith("Fatal error"))
        {
            Tester.errorLogger.AddError(
                "The weaving was unsuccesful, when weaving target: '" + target
                    + "' " +
                "with aspect: '" + aspect + "', by pointcut: '" + pointcutFile
                    + "', " +
                "and with outputfile: '" + outputFile + "'.\n" +
                "This is the output from the weaver: \n" + output);
            return false;
        }
        return true;
    }

    /// <summary>
    /// Checks if the console output of the weaving was as expected.
    /// </summary>
    /// <param name="output">The console output from the weaving.</param>
    /// <param name="target">Target of the weaving.</param>
    /// <param name="aspect">Aspect of the weaving.</param>
    /// <param name="pointcutFile">The pointcut file to weave after.</param>
    /// <param name="outputFile">The name of the file which might have been
        outputted by the weaving.</param>
    /// <param name="expectedOutput">The expected output to check up against
        .</param>
    /// <returns>A boolean indicating if the output matched the expected
        output.</returns>
    private static bool checkWeaverExpectedOutput(string output, string target
        , string aspect, string pointcutFile, string outputFile, string
        expectedOutput)
    {
        if (output.Equals(expectedOutput, StringComparison.
            CurrentCultureIgnoreCase))
            return true;

        Tester.errorLogger.AddError(
                "The weaving was expected to be unsuccesful, when weaving
                    target: '" + target + "' " +
```

```
                "with aspect: '" + aspect + "', by pointcut: '" + pointcutFile
                    + "', " +
                "and with outputfile: '" + outputFile + "'.\n" +
                "But the output was not as expected.\n"+
                "This is the output from the weaver: \n" + output +"\nWhile
                    the expected output was:\n"+expectedOutput);
            return false;
    }

    /// <summary>
    /// Calls a method in the new assembly outputted by the weaver, and checks
         if the returned object is as expected.
    /// </summary>
    /// <param name="expectedReturnValue">The value/object expected to be
        returned from the call.</param>
    /// <param name="className">The name of the class in which the method is
        located.</param>
    /// <param name="methodName">The name of the method to call.</param>
    /// <param name="args">An object array with the arguments to the method
        call.</param>
    public static void ExpectReturnOnCall(object expectedReturnValue, string
        className, string methodName, object[] args)
    {
        string outputFilename = System.IO.Directory.GetCurrentDirectory() + "/
            outputFiles/" + getCallerIDName() + "/" + getCallerIDName() + "Out
            ";
        if (File.Exists(outputFilename + ".dll")) //find out if the outputted
            file is an exe or dll file.
            outputFilename += ".dll";
        else
            outputFilename += ".exe";

        Assembly assembly = Assembly.LoadFile(outputFilename);
        foreach (Type type in assembly.GetTypes())
        {
            if (!type.Name.Equals(className))
                continue;

            // If it is the right class, find the method.
            object testClass = Activator.CreateInstance(type);
            object returnValue;
            try
            {
                returnValue = type.InvokeMember(methodName, BindingFlags.
                    NonPublic | BindingFlags.Instance | BindingFlags.Static |
                    BindingFlags.Public | BindingFlags.InvokeMethod, null,
                    testClass, args);
            }
            catch (System.MissingMethodException e)
            {
                Tester.errorLogger.AddError(e.Message + "\n" +
                    "This happend in the test method: '" + getCallerID() + "'.
                        ");
                assembly = null;
                return;
            }
            catch (Exception e)
            {
                Tester.errorLogger.AddError("While running '
                    ExpectedRuturnValue' from test method: '" + getCallerID()
                    + "', the following error was thrown:\n" + e.ToString());
                assembly = null;
```

```
                return;
            }
            assembly = null;

            if ((expectedReturnValue == null && returnValue == null) ||
                expectedReturnValue.Equals(returnValue))
                return;



            Tester.errorLogger.AddError(
                "The expected value from the call was: '" +
                    expectedReturnValue + "' - type: '" + expectedReturnValue.
                    GetType() + "',\n" +
                "but the returned value from the call to '" + className + ":"
                    + methodName + "' " +
                "was '" + returnValue + "' - type: '" + returnValue.GetType()
                    + "'.\n" +
                "This happend in the test method: '" + getCallerID() + "'.");
        }
    }

    /// <summary>
    /// Gets the ID of the method that has called the method which calls the
        method. (That is two calls back).
    /// </summary>
    /// <returns>A string with the type:name of the calling method.</returns>
    private static string getCallerID()
    {
        //Getting the caller ID
        System.Diagnostics.StackFrame sf = new System.Diagnostics.StackFrame
            (2);
        System.Reflection.MethodBase mb = sf.GetMethod();
        //string assemblyName = mb.DeclaringType.Assembly.GetName().Name;
        return mb.DeclaringType.Name + ":" + mb.Name;
    }

    /// <summary>
    /// Gets the name of last method in the call tree, that doesn't start with
         "Weave".
    /// </summary>
    /// <returns>The name of the method.</returns>
    private static string getCallerIDName()
    {
        //Getting the caller ID
        int i = 2;
        System.Reflection.MethodBase mb;
        do
        {
            System.Diagnostics.StackFrame sf = new System.Diagnostics.
                StackFrame(i);
            mb = sf.GetMethod();
            i++;
        }
        while (mb.Name.Equals("Weave"));
        return mb.Name;
    }

}
}
```

TestableClassAttribute.cs - Test class indicator attribute

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAWTester
{

    /// <summary>
    /// Indicator attribute used to indicate that a class contains test methods.
    /// </summary>
    [AttributeUsage(AttributeTargets.Class)]
    public class TestableClassAttribute : Attribute
    {
    }
}
```

TestableMethodAttribute.cs - Test method indicator attribute

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAWTester
{
    /// <summary>
    /// An indicator attribute, that shows that a method is a test method
    /// </summary>
     [AttributeUsage(AttributeTargets.Method)]
    public class TestableMethodAttribute : Attribute
    {
        private string _inputFile;
        private bool _hasOutputFile;

        public string InputFile
        { get { return _inputFile; } }

        public bool HasOutputFile
        { get { return _hasOutputFile; } }

        /// <summary>
        /// Shows that a method is a test method.
        /// </summary>
        /// <param name="inputfile">The path and name of the inputfile which the
            weaver will use.</param>
        public TestableMethodAttribute(string inputfile)
        {
            _inputFile = inputfile;
            _hasOutputFile = true;
        }

        /// <summary>
        /// Shows that a method is a test method.
        /// </summary>
        /// <param name="inputfile">The path and name of the inputfile which the
            weaver will use.</param>
        /// <param name="hasOutputFile">Indication whether the weaving is expected
             to create an output file.</param>
        public TestableMethodAttribute(string inputfile, bool hasOutputFile)
        {
```

```
            _inputFile = inputfile;
            _hasOutputFile = hasOutputFile;
        }
    }
}
```

tests.cs - the tests

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAWTester;

namespace Target
{
    [TestableClass()]
    public class Tests
    {
        /* Test 1
         * Test of:
         * In advice: Return defined primitive - without proceed.
         * In advice: Use of opcode "box".
         * Pointcut interception: Access specifier:              Public
         * Pointcut interception: Returntype:                    Primitive
         * Pointcut interception: Type name specification:       *.name
         * Pointcut interception: Method name specification:     Name
         * Pointcut interception: Matching advice methods:       One
         * Pointcut interception: Matching targets:              One
         */
        [TestableMethod("../../../target/bin/Release/Target.dll")]
        private void Test1()
        {
            API.CreatePointcutFile("around public * int *.Interception.Target:
                Test1() do Aspect.Aspect:Test1Aspect;", "pointcutfile");
            API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect/
                bin/Release/Aspect.dll", "pointcutfile");
            API.ExpectReturnOnCall(42, "Target", "Test1", null);
        }

        /* Test 2.
         * Test of:
         * In advice: Return defined string - without proceed.
         * Pointcut interception: Returntype:                    *
         */
        [TestableMethod("../../../target/bin/Release/Target.dll")]
        private void Test2()
        {
            API.CreatePointcutFile("around public * * *.Interception.Target:Test2(
                string) do Aspect.Aspect:Test2Aspect;", "pointcutfile");
            API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect/
                bin/Release/Aspect.dll", "pointcutfile");
            object[] args = new object[1];
            args[0] = "Hello";
            API.ExpectReturnOnCall("HelloHello", "Target", "Test2", args);
        }

        /* Test 3.
         * Test of:
         * In advice: Return defined string - with proceed.
         * Pointcut interception: Returntype:                    Fully defined
         * Pointcut interception: Type name specification:       Name.name
```

```
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test3()
{
    API.CreatePointcutFile("around public static string Target.
        Interception.Target:Test3(*) do Aspect.Aspect:Test3Aspect;", "
        pointcutfile");
    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect/
        bin/Release/Aspect.dll", "pointcutfile");
    string arg0 = "Hello World";
    object[] args = new object[1];
    args[0] = arg0;
    API.ExpectReturnOnCall(arg0, "Target", "Test3", args);
}


/* Test 4.
 * Test of:
 * In advice: Return defined external - with proceed.
 * In advice: Use of opcode "call".
 * Pointcut interception: Access specification:          *
 * Pointcut interception: Invocation type:               *
 * Pointcut interception: Method name specification:     Name*
 * Pointcut interception: Argument definition:           *
 * Pointcut interception: Matching advice methods:       More than one
 * Pointcut interception: Matching targets:              More than one
 */
[TestableMethod("../../../target2/bin/Release/Target2.dll")]
private void Test4()
{
    API.CreatePointcutFile(
        "around * * MyTestLib.MyType *.Test4.Test4Target:Test4*(*) do
            Aspect.Aspect:Test4Aspect;",
        "pointcutfile");
    API.Weave("../../../Target2/bin/Release/Target2.dll", "../../../Aspect
        /bin/Release/Aspect.dll", "pointcutfile");
    MyTestLib.MyType expectedReturn1 = new MyTestLib.MyType();
    API.ExpectReturnOnCall(expectedReturn1, "Target2", "Test4A", null);

    MyTestLib.MyType expectedReturn2 = new MyTestLib.MyType();
    string arg0 = "Hello";
    expectedReturn2.Name = arg0 + " " + arg0;
    object[] args = new object[1];
    args[0] = arg0;
    API.ExpectReturnOnCall(expectedReturn2, "Test4Target", "Test4B", args)
        ;

    MyTestLib.MyType expectedReturn3 = new MyTestLib.MyType();
    string arg1 = "World";
    expectedReturn3.Name = arg0 + " " + arg1 + " " + arg0;
    args = new object[2];
    args[0] = arg0;
    args[1] = arg1;
    API.ExpectReturnOnCall(expectedReturn3, "Test4Target", "Test4C", args)
        ;
}

/* Test 5.
 * Test of:
 * In advice: Return defined void - with proceed.
 * Pointcut interception: Target which doesn't match:   On returntype
 */
```

```
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test5()
{
    API.CreatePointcutFile("around public * * *.Interception.Target:Test5
        *() do Aspect.Aspect:Test5Aspect;", "pointcutfile");
    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect/
        bin/Release/Aspect.dll", "pointcutfile");
    API.ExpectReturnOnCall(null, "Target", "Test5A", null);
    API.ExpectReturnOnCall(42, "Target", "Test5B", null);
}

/* Test 6.
 * Test of:
 * In advice: Return defined external - without proceed.
 * In advice: Use of opcode: Access an argument.
 * Pointcut interception: Type name specification:      *
 * Pointcut interception: Method name specification:    *
 * Pointcut interception: Argument definition:          Defined
 */
[TestableMethod("../../../target2/bin/Release/Target2.dll")]
private void Test6()
{
    API.CreatePointcutFile("around * * MyTestLib.MyType *:*(MyTestLib.
        MyType) do Aspect.Aspect:Test6Aspect;", "pointcutfile");
    API.Weave("../../../Target2/bin/Release/Target2.dll", "../../../Aspect
        /bin/Release/Aspect.dll", "pointcutfile");
    MyTestLib.MyType expectedReturn = new MyTestLib.MyType();
    expectedReturn.Name = "Aspect";
    object[] args = new object[1];
    args[0] = new MyTestLib.MyType();
    API.ExpectReturnOnCall(expectedReturn, "Test6Target", "Test6", args);

}

/* Test 7.
 * Test of:
 * In advice: Return defined aspect - without proceed.
 * - (not possible, as aspect is not introduced into the target assembly)
 * Pointcut interception: Access specification:         Private
 * Pointcut interception: Type name specification:      Name.*
 * Pointcut interception: Method name specification:    *name
 */
[TestableMethod("../../../target3/bin/Release/Target3.dll",false)]
private void Test7()
{
    string expectedOutput = "Fatal error: Unable to instantiate type '
        AspectType' from 'Test7', as 'AspectType' is not defined in the
        target assembly. If this type should be available in the target
        assembly, please specify that it should be inserted into the
        target assembly using the pointcut specification.\r\n";
    API.CreatePointcutFile("around private * * Target.*:*7() do Aspect.
        Aspect:Test7Aspect;", "pointcutfile");
    API.WeaveWithExpectedOutput("../../../Target3/bin/Release/Target3.dll"
        , "../../../Aspect/bin/Release/Aspect.dll", "pointcutfile",
        expectedOutput);
}

/* Test 8.
 * Test of:
 * In advice: Return defined target - without proceed.
 * Pointcut interception: Access specification:         Internal
 */
```

```
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test8()
{
    API.CreatePointcutFile("around internal * * *:*8(string) do Aspect.
        Aspect:Test8Aspect;", "pointcutfile");
    API.Weave("../../../Target/bin/Release/Target.dll", "../../../Aspect/
        bin/Release/Aspect.dll", "pointcutfile");
    //Not possible to test, as the Target.Interception.Target will be of a
         different kind
    //when using the new outputfile.
    /*Target.Interception.Target expextedOutput = new Target.Interception.
        Target();
    expextedOutput.testValue = "Aspect";
    object[] args = new object[1];
    args[0] = "test";
    API.ExpectReturnOnCall(expextedOutput, "Target", "Test8", args);*/
}

/* Test 9.
 * Test of:
 * In advice: Return defined void - without proceed.
 * Pointcut interception: Access specification:          Protected
 * Pointcut interception: Invocation type:               Static
 * Pointcut interception: Argument definition:           Defined as
     primitive
 * Pointcut interception: Target which doesn't match:    On argument type
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test9()
{
    API.CreatePointcutFile("around protected static void *.Interception.
        Target:Test9*(int) do Aspect.Aspect:Test9;", "pointcutfile");
    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect/
        bin/Release/Aspect.dll", "pointcutfile");
    object[] args = new object[1];
    args[0] = 42;
    API.ExpectReturnOnCall(null, "Target", "Test9", args);
}


/* Test 10.
 * Test of:
 * In advice: Use of opcode: Accessing local variable.
 * In advice: Use of opcode: newobj.
 * Introduction: Class with reference to:                Aspect Assembly
 * Introduction: Method with reference to:               Aspect Assembly
 * Pointcut introduction: Access specifier (method):     Public
 * Pointcut introduction: Access specifier (method):     *
 * Pointcut introduction: Invocation type (method):      *
 * Pointcut introduction: Returntype (method):           Mscorlib
 * Pointcut introduction: Target (method):               xxx.yyy
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test10()
{
    API.CreatePointcutFile(
        "insert class Aspect2.Test10InsertClass into Target.Test10;" +
        "around public * * *:Test10*(*) do Aspect2.Test10Class:
            Test10Aspect;" +
        "insert method * * * Aspect2.Test10Class:Test10Int() into Target.
            Test10.Test10Target;" +
        "insert method private * string Aspect2.Test10Class:Test10String(
```

```
            int) into Target.Test10.Test10Target;",
        "pointcutfile");
    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
        bin/Release/Aspect2.dll", "pointcutfile");
    API.ExpectReturnOnCall(42, "Target4", "Test10A", null);
    object[] args = new object[1];
    args[0] = 42;
    API.ExpectReturnOnCall("Aspect" + args[0], "Test10Target", "Test10B",
        args);
}


/* Test 11.
 * Test of:
 * In advice: Return generic type (void) - with proceed.
 * Pointcut interception: Invocation type:              Instance
 * Pointcut introduction: Returntype (method):          Void
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test11()
{
    API.CreatePointcutFile(
        "around public * * *.Test11Target:*(*) do Aspect2.Test11Class:
            Test11Aspect;" +
        "insert method public instance void Aspect2.Test11Class:
            Test11Method(string) into Target.Test11.Test11Target;",
    "pointcutfile");
    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
        bin/Release/Aspect2.dll", "pointcutfile");
    API.ExpectReturnOnCall(42, "Test11Target", "Test11", null);
}


/* Test 12.
 * Test of:
 * In advice: Return defined primitive - with proceed.
 * In advice: Use of opcode: unbox.
 * Introduction: Class with reference to:               Mscorlib
 * Introduction: Method with reference to:              Mscorlib
 * Introduction: Field with reference to:               Mscorlib
 * Introduction: Property with reference to:            Mscorlib
 * Introduction: Event with reference to:               Mscorlib
 * Introduction: Property with reference to:            Aspect Assembly
 * Pointcut introduction: Access specifier (property):  Public
 * Pointcut introduction: Access specifier (event):     Public
 * Pointcut introduction: Access specifier (field):     Internal
 * Pointcut introduction: Invocation type (method):     Instance
 * Pointcut introduction: Invocation type (event):      Instance
 * Pointcut introduction: Invocation type (field):      *
 * Pointcut introduction: Invocation type (property):   *
 * Pointcut introduction: Returntype (field):           Mscorlib
 * Pointcut introduction: Returntype (property):        Mscorlib
 * Pointcut introduction: Returntype (event):           Mscorlib
 * Pointcut introduction: Target (field):               xxx.yyy
 * Pointcut introduction: Target (property):            xxx.yyy
 * Pointcut introduction: Target (class):               xxx.yyy
 * Pointcut introduction: Target (event):               xxx.yyy
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test12()
{
    API.CreatePointcutFile(
```

```
            "around public instance int *:Test12(int) do Aspect2.Test12Class:
                Test12Aspect;" +
            "insert method public instance int Aspect2.Test12Class:
                Test12Return3() into Target.Test12.Test12Target;" +
            "insert class Aspect2.Test12InsertClass into Target.Test12.
                Test12Target;" +
            "insert field internal * int Aspect2.Test12Class:a into Target.
                Test12.Test12Target;" +
            "insert property public * int Aspect2.Test12Class:IntField into
                Target.Test12.Test12Target;" +
            "insert event public instance System.EventHandler Aspect2.
                Test12Class:Test12 into Target.Test12.Test12Target;"
            , "pointcutfile");
        API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
            bin/Release/Aspect2.dll", "pointcutfile");
        object[] args = new object[1];
        args[0] = 40;
        API.ExpectReturnOnCall(40, "Test12Target", "Test12", args);

    }

    /* Test 13.
     * Test of:
     * In advice: Use of opcode: Accessing a field.
     * Introduction: Class with reference to:              Target assembly
     * Introduction: Method with reference to:             Target assembly
     * Introduction: Field with reference to:              Target assembly
     * Introduction: Property with reference to:           Target assembly
     * Introduction: Event with reference to:              Target assembly
     * Pointcut introduction: Access specifier (field):    Private
     * Pointcut introduction: Access specifier (event):    Private
     * Pointcut introduction: Access specifier (method):   Internal
     * Pointcut introduction: Access specifier (Property): Internal
     * Pointcut introduction: Invocation type (field):     Static
     * Pointcut introduction: Invocation type (event):     Static
     * Pointcut introduction: Invocation type (property):  instance
     * Pointcut introduction: Returntype (field):          Target assembly
     * Pointcut introduction: Returntype (property):       Target assembly
     * Pointcut introduction: Returntype (event):          Target assembly
     * Pointcut introduction: Target (method):             *.xxx
     * Pointcut introduction: Target (field):              *.xxx
     * Pointcut introduction: Target (property):           *.xxx
     * Pointcut introduction: Target (event):              *.xxx
     */
    [TestableMethod("../../../target/bin/Release/Target.dll")]
    private void Test13()
    {
        API.CreatePointcutFile(
            "around public instance * *:ReturnITest13(*) do Aspect2.
                Test13Class:Test13Aspect;" +
            "insert field private static Target.Test13.Test13Target Aspect2.
                Test13Class:target13 into *.Test13Target;" +
            "insert method * * System.EventHandler Aspect2.Test13Class:
                target13_Test13Event() into *.Test13Target;" +
            "insert property internal instance Target.Test13.Test13Target
                Aspect2.Test13Class:Target13 into *.Test13Target;" +
            "insert class Aspect2.Test13Class.Test13NestedClass into Target.
                Test13.Test13Target;" +
            "insert event private static Target.Test13.Test13EventHandler
                Aspect2.Test13Class:Test13AspectEvent into *.Test13Target;",
            "pointcutfile");
        API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
```

```
            bin/Release/Aspect2.dll", "pointcutfile");
        //Not possible to test, as the Target.Test13.ITest13 will be of a
            different kind
        //when using the new outputfile.
        /*
        object[] args = new object[1];
        args[0] = new Target.Test13.Test13Target();
        Target.Test13.Test13Target expected = new Target.Test13.Test13Target()
            ;
        expected.testField = 4;
        API.ExpectReturnOnCall(expected, "Test13Target", "ReturnITest13", args
            );*/

    }

    /* Test 14.
     * This is a special test, as it has been created to test the opcode "
        ldtoken".
     * To get the opcode into a program, it was necessary to insert it
        manually by writeting the needed Assembler IL codes
     * into an a basic assembly. There is therefor no aspect source code for
        this test.
     * The IL for the test looks as follows:

            nop
            ldtoken     TempAspect.Test14
            pop
            ldtoken     method instance void TempAspect.Class1::Test14A()
            pop
            ldtoken     field int32 TempAspect.Class1::test14field
            pop
            ret

     * So the test it testing with each kind of tokentype usable for this
        opcode.
     * The correctness of test is checked by peverify (automaticly) and by
        usage of ildasm.
     */
    [TestableMethod("../../../target/bin/Release/Target.dll")]
    private void Test14()
    {
        API.CreatePointcutFile(
        "around public instance int Target.Test14.Test14Target:Test14(*) do
            Aspect.Test14Class:Test14Aspect;" +
        "insert method public instance void Aspect.Test14Class:Test14A() into
            Target.Test14.Test14Target;" +
        "insert field public * int Aspect.Test14Class:test14field into Target.
            Test14.Test14Target;" +
        "insert class Aspect.Test14 into Target.Test14;" +
        "insert method public instance int Aspect.Test14Class:Test14Aspect()
            into Target.Test14.Test14Target;",
        "pointcutfile");

        API.Weave("../../../target/bin/Release/Target.dll", "specialtest/out.
            dll", "pointcutfile");
    }

    /* Test 15.
     * Test of:
     * In advice: Return defined Target - with return.
     * Introduction: Field with reference to:                Aspect assembly
     * Pointcut interception: Inherit defined.
```

```
 * Pointcut introduction: Access specfier (field):      Public
 * Pointcut introduction: Invocation type (field):      Instance
 * Pointcut introduction: Target (field):               xxx.*
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test15()
{
    API.CreatePointcutFile(
        "around public instance * Target.Test15.Test15Target:Test15(*)
            inherits Target4.Test15I do Aspect2.Test15Class:Test15;" +
        "insert field public instance string Aspect2.Test15Class:
            aspectValue into Target.Test15.*;",
        "pointcutfile");
    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
        bin/Release/Aspect2.dll", "pointcutfile");
}

/* Test 16.
 * Test of:
 * Introduction: Method with reference to:              External assembly
 * Introduction: Field with reference to:               External assembly
 * Introduction: Property with reference to:            External assembly
 * Introduction: Class with reference to:               External assembly
 * Introduction: Event with reference to:               External assembly
 * Pointcut introduction: Access specifier (method):    Protected
 * Pointcut introduction: Access specifier (field):     Protected
 * Pointcut introduction: Access specifier (property):  Protected
 * Pointcut introduction: Access specifier (event):     Protected
 * Pointcut introduction: Invocation type (method):     Static
 * Pointcut introduction: Invocation type (property):   Static
 * Pointcut introduction: Returntype (method):          External assembly
 * Pointcut introduction: Returntype (field):           External assembly
 * Pointcut introduction: Returntype (property):        External assembly
 * Pointcut introduction: Returntype (event):           External assembly
 * Pointcut introduction: Target (method):              *
 * Pointcut introduction: Target (field):               *
 * Pointcut introduction: Target (property):            *
 * Pointcut introduction: Target (class):               *
 * Pointcut introduction: Target (event):               *
 */
[TestableMethod("../../../target2/bin/Release/Target2.dll")]
private void Test16()
{
    API.CreatePointcutFile(
        "around public instance * *.Test16.Test16Target:Test16() do
            Aspect2.Test16Class:Test16Aspect;" +
        "insert method protected static MyTestLib.Test16External Aspect2.
            Test16Class:TestMethod() into *;" +
        "insert method protected * * Aspect2.Test16Class:
            Test16ClassTestEvent() into *;" +
        "insert property protected static MyTestLib.Test16External Aspect2
            .Test16Class:TestProperty into *;" +
        "insert field protected static MyTestLib.Test16External Aspect2.
            Test16Class:testField into *;" +
        "insert event protected static MyTestLib.Test16EventHandler
            Aspect2.Test16Class:TestEvent into *;" +
        "insert class Aspect2.Test16Class.Test16NestedClass into *;",
        "pointcutfile");
    API.Weave("../../../target2/bin/Release/Target2.dll", "../../../
        Aspect2/bin/Release/Aspect2.dll", "pointcutfile");
    MyTestLib.Test16External t = new MyTestLib.Test16External();
    t.value = "Aspect";
```

```
        API.ExpectReturnOnCall(t, "Test16Target", "Test16", null);
    }

    /* Test 17.
     * Test of:
     * In advice: Return generic type (primitive) - with proceed.
     * In advice: Return generic type (string) - with proceed.
     * In advice: Return generic type (external) - with proceed.
     * In advice: Return generic type (target) - with proceed.
     */
    [TestableMethod("../../../target2/bin/Release/Target2.dll")]
    private void Test17()
    {
        API.CreatePointcutFile(
            "around public instance * *.Test17Target:Test17(*) do Aspect.
                Aspect:Test17Aspect;",
            "pointcutfile");

        API.Weave("../../../target2/bin/Release/Target2.dll", "../../../Aspect
            /bin/Release/Aspect.dll", "pointcutfile");
        object[] args = new object[1];
        args[0] = new MyTestLib.Test16External();
        API.ExpectReturnOnCall(args[0], "Test17Target", "Test17", args);
        args[0] = "Hello";
        API.ExpectReturnOnCall(args[0], "Test17Target", "Test17", args);
        args[0] = 10;
        API.ExpectReturnOnCall(args[0], "Test17Target", "Test17", args);
        // Not testable, as Test17Target here and in the new target assembly
            is not the same.
        //args[0] = new Target.Test17.Test17Target();
        //API.ExpectReturnOnCall(args[0], "Test17Target", "Test17", args);

    }

    /* Test 18.
     * Test of:
     * In advice: Use of the opcode "initobj".
     * In advice: Return generic type (primitive) - using default with proceed
         .
     * In advice: Return generic type (string) - using default with proceed.
     * In advice: Return generic type (external) - using default with proceed.
     * In advice: Return generic type (target) - using default with proceed.
     * In advice: Return generic type (void) - using default with proceed.
     * As the use of the default method on the mentioned types generates 'null
         ',
     * this is tested, and furthermore the methods prints out the string "
         Aspect"
     * on the console.
     */
    [TestableMethod("../../../target2/bin/Release/Target2.dll")]
    private void Test18()
    {
        API.CreatePointcutFile(
            "around public instance * *.Test18Target:Test18*() do Aspect.
                Aspect:Test18Aspect;",
            "pointcutfile");

        API.Weave("../../../target2/bin/Release/Target2.dll", "../../../Aspect
            /bin/Release/Aspect.dll", "pointcutfile");
        API.ExpectReturnOnCall(0, "Test18Target", "Test18int", null);
        API.ExpectReturnOnCall(null, "Test18Target", "Test18string", null);
        API.ExpectReturnOnCall(null, "Test18Target", "Test18", null);
```

```
    API.ExpectReturnOnCall(null, "Test18Target", "Test18target", null);
    API.ExpectReturnOnCall(null, "Test18Target", "Test18external", null);
}

/* Test 19.
 * Test of:
 * In advice: Return generic type (primitive) - using default without
     proceed.
 * In advice: Return generic type (string) - using default without proceed
     .
 * In advice: Return generic type (external) - using default without
     proceed.
 * In advice: Return generic type (target) - using default without proceed
     .
 * In advice: Return generic type (void) - using default without proceed.
 * As the use of the default method on the mentioned types generates 'null
     ',
 * this is tested, and furthermore the methods prints out the string "
     Aspect"
 * on the console.
 */
[TestableMethod("../../../target2/bin/Release/Target2.dll")]
private void Test19()
{
    API.CreatePointcutFile(
        "around public instance * *.Test18Target:Test18*() do Aspect.
            Aspect:Test19Aspect;",
        "pointcutfile");

    API.Weave("../../../target2/bin/Release/Target2.dll", "../../../Aspect
        /bin/Release/Aspect.dll", "pointcutfile");
    API.ExpectReturnOnCall(0, "Test18Target", "Test18int", null);
    API.ExpectReturnOnCall(null, "Test18Target", "Test18string", null);
    API.ExpectReturnOnCall(null, "Test18Target", "Test18", null);
    API.ExpectReturnOnCall(null, "Test18Target", "Test18target", null);
    API.ExpectReturnOnCall(null, "Test18Target", "Test18external", null);
}


/* Test 20.
 * Test of:
 * In advice: Use of opcode "newarr".
 * Pointcut introduction: Access specifier (method):     Private
 * Pointcut introduction: Access specifier (property):   Private
 * Pointcut introduction: Access specifier (field):      *
 * Pointcut introduction: Access specifier (property):   *
 * Pointcut introduction: Access specifier (event):      *
 * Pointcut introduction: Access specifier (event):      Internal
 * Pointcut introduction: invocation type (event):       *
 * Pointcut introduction: Returntype (method):           Target assembly
 * Pointcut introduction: Returntype (method):           Aspect assembly
 * Pointcut introduction: Returntype (field):            Aspect assembly
 * Pointcut introduction: Returntype (property):         Aspect assembly
 * Pointcut introduction: Target (method):               xxx.*
 * Pointcut introduction: Target (property):             xxx.*
 * Pointcut introduction: Target (class):                *.xxx
 * Pointcut introduction: Target (event):                xxx.*
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test20()
{
    API.CreatePointcutFile(
```

```
            "around public instance string *:Test20() do Aspect2.Test20Class:
                Test20Aspect;" +
            "insert method private * Aspect2.Test20InsertClass Aspect2.
                Test20Class:Test20AspectReturn() into Target.Test20.
                Test20Target;" +
            "insert class Aspect2.Test20InsertClass into *.Test20;" +
            "insert field * * Aspect2.Test20InsertClass Aspect2.Test20Class:
                test20Field into Target.Test20.*;" +
            "insert method * * Target.Test20.Test20Target Aspect2.Test20Class:
                Test20targetReturn() into Target.Test20.*;" +
            "insert event internal * * Aspect2.Test20Class:Test20Event  into
                Target.Test20.*;" +
            "insert method * * * Aspect2.Test20Class:Test20ClassTest20Event(
                object, System.EventArgs) into Target.Test20.Test20Target;" +
            "insert event * * * Aspect2.Test20Class:Test20Event2 into Target.
                Test20.Test20Target;" +
            "insert method * * * Aspect2.Test20Class:Test20ClassTest20Event2(
                object, System.EventArgs) into Target.Test20.Test20Target;" +
            "insert property private * * Aspect2.Test20Class:
                Test20StringProperty into Target.Test20.*;" +
            "insert property * * Aspect2.Test20InsertClass Aspect2.Test20Class
                :Test20Property into Target.Test20.*;",
            "pointcutfile");

        API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
            bin/Release/Aspect2.dll", "pointcutfile");
        API.ExpectReturnOnCall("Test20 aspect aspect", "Test20Target", "Test20
            ", null);
    }

    /* Test 21.
     * Test of:
     * Modification: Change of basetype.
     * Pointcut modification: Action:                    Inherit
     * Pointcut introduction: Target (class):            xxx.*
     */
    [TestableMethod("../../../target/bin/Release/Target.dll")]
    private void Test21()
    {
        API.CreatePointcutFile(
            "insert class Aspect2.Test21Class into Target.Test21.*;" +
            "modify Target.Test21.ns.Test21Target inherit Aspect2.Test21Class;
                ",
            "pointcutfile");

        API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
            bin/Release/Aspect2.dll", "pointcutfile");
        API.ExpectReturnOnCall("Aspect", "Test21Target", "Test21Method", null)
            ;
    }


    /* Test 22.
     * Test of:
     * Modification: Implementation of interface.
     * Pointcut modfication: Action:                     Implement
     */
    [TestableMethod("../../../target/bin/Release/Target.dll")]
    private void Test22()
    {
        API.CreatePointcutFile(
            "insert method * * * Aspect2.Test22Class:Test22IMethod() into
```

```
            Target.Test22.*;" +
        "modify Target.Test22.Test22Target implement Aspect2.
            Test22Interface;" +
        "insert class Aspect2.Test22Interface into Target.Test22;" +
        "around * * * *:Test22Method() do Aspect2.Test22Class:Test22Aspect
            ;",
        "pointcutfile");

    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
        bin/Release/Aspect2.dll", "pointcutfile");
    API.ExpectReturnOnCall("Aspect", "Test22Target", "Test22Method", null)
        ;
}

/* Test 23.
 * Test of:
 * In advice: JoinPoint API: Proceed<T>
 * In advice: JoinPoint API: GetTarget<T>
 * In advice: JoinPoint API: AccessSpecfier
 * In advice: JoinPoint API: Arguments
 * In advice: JoinPoint API: DeclaringType
 * In advice: JoinPoint API: IsStatic
 * In advice: JoinPoint API: Name
 * In advice: JoinPoint API: ReturnType
 * In advice: Use of opcode "switch"
 * The switch:

        case 0:
            return JoinPointContext.Proceed<string>();
        case 1:
            return JoinPointContext.AccessSpecifier;
        case 2:
            return JoinPointContext.Arguments;
        case 3:
            return JoinPointContext.DeclaringType;
        case 4:
            return JoinPointContext.IsStatic.ToString();
        case 5:
            return JoinPointContext.Name;
        case 6:
            return JoinPointContext.ReturnType;
        case 7:
            return JoinPointContext.GetTarget<Target5.T23.T23Target>().
                ToString();
        default:
            return "Aspect";
 */
[TestableMethod("../../../target/bin/Release/Target.dll")]
private void Test23()
{
    API.CreatePointcutFile(
        "around * * * *:Test23Method(*) do Aspect2.Test23Class:
            Test23Aspect;",
        "pointcutfile");

    API.Weave("../../../target/bin/Release/Target.dll", "../../../Aspect2/
        bin/Release/Aspect2.dll", "pointcutfile");
    object[] args = new object[1];
    args[0] = 0;
    API.ExpectReturnOnCall("0", "Test23Target", "Test23Method", args);
    args[0] = 1;
    API.ExpectReturnOnCall("public", "Test23Target", "Test23Method", args)
```

```
            ;
        args[0] = 2;
        API.ExpectReturnOnCall("System.Int32", "Test23Target", "Test23Method",
            args);
        args[0] = 3;
        API.ExpectReturnOnCall("Target.Test23.Test23Target", "Test23Target", "
            Test23Method", args);
        args[0] = 4;
        API.ExpectReturnOnCall("False", "Test23Target", "Test23Method", args);
        args[0] = 5;
        API.ExpectReturnOnCall("Test23Method", "Test23Target", "Test23Method",
            args);
        args[0] = 6;
        API.ExpectReturnOnCall("System.String", "Test23Target", "Test23Method"
            , args);
        args[0] = 7;
        API.ExpectReturnOnCall("TargetToString", "Test23Target", "Test23Method
            ", args);
        args[0] = 8;
        API.ExpectReturnOnCall("Aspect", "Test23Target", "Test23Method", args)
            ;
    }

  }
}
```

Aspect.cs - Aspect file for interception tests only

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAW.API;

namespace Aspect
{
    /// <summary>
    /// This class contains the aspects for all the tests that only uses
        interception
    /// </summary>
    public class Aspect
    {
        /* Test 1.
         * Test of:
         * Return defined primitive - without proceed
         * Use of opcode "box" (a is boxed, to be used in the string)
         */
        public int Test1Aspect()
        {
            int a = 42;
            Console.WriteLine("Test 1 - advice. The output is: {0}", a);
            return a;
        }

        /*************************************************************************
            */

        /* Test 2.
         * Return defined string - without proceed
         */
        public string Test2Aspect(string a)
        {
```

```
    return a + a;
}

/*************************************************************************
    */

/* Test 3.
 * Test of:
 * Return defined string - with proceed
 */
public static string Test3Aspect()
{
    return JoinPointContext.Proceed<string>();
}

/*************************************************************************
    */

#region Test4

/* Test 4.
 * Test of:
 * Return defined external - with proceed
 * Use of opcode "call"
 */
public MyTestLib.MyType Test4Aspect()
{
    MyTestLib.MyType returnvalue = JoinPointContext.Proceed<MyTestLib.
        MyType>();
    Console.WriteLine(returnvalue.Name);
    return returnvalue;
}

public MyTestLib.MyType Test4Aspect(string a)
{
    MyTestLib.MyType returnvalue = JoinPointContext.Proceed<MyTestLib.
        MyType>();
    returnvalue.Name += " " + a;
    Console.WriteLine(returnvalue.Name);
    return returnvalue;
}

#endregion

/*************************************************************************
    */

/* Test 5.
 * Test of:
 * Return defined void - with proceed
 */
public void Test5Aspect()
{
    JoinPointContext.Proceed<YIIHAW.API.Void>();
    return;
}

/*************************************************************************
    */

/* Test 6.
 * Test of:
```

```
 * Return defined external - without proceed
 * use of opcode: Access an argument
 */
public MyTestLib.MyType Test6Aspect(MyTestLib.MyType m)
{
    m.Name = "Aspect";
    return m;
}

/*****************************************************************
    */

/* Test 7.
 * Test of:
 * Return defined aspect - without proceed
 * (not possible, as aspect is not introduced into the target assembly)
 */
private AspectType Test7Aspect()
{
    return new AspectType();
}

/*****************************************************************
    */

/* Test 8.
 * Test of:
 * Return defined target - without proceed
 */
internal Target.Interception.Target Test8Aspect()
{
    Target.Interception.Target t = new Target.Interception.Target();
    t.testValue = "Aspect";
    return t;
}

/*****************************************************************
    */

/* Test 9.
 * Test of:
 * Return defined void - without proceed
 */
protected static void Test9Aspect(int i)
{
    Console.WriteLine("Aspect" + i);
}

/*****************************************************************
    */

/* Test 17.
 * Test of:
 * Return generic type (primitive) - with proceed
 * Return generic type (string) - with proceed
 * Return generic type (external) - with proceed
 * Return generic type (target) - with proceed
 */
public T Test17Aspect<T>()
{
    Console.WriteLine("Aspect");
    return JoinPointContext.Proceed<T>();
```

```
        }

        /*************************************************************************
            */

        /* Test 18.
         * Test of:
         * Use of the opcode "initobj".
         * Return generic type (primitive) - using default with proceed
         * Return generic type (string) - using default with proceed
         * Return generic type (external) - using default with proceed
         * Return generic type (target) - using default with proceed
         * Return generic type (void) - using default with proceed
         */
        public T Test18Aspect<T>()
        {
            Console.WriteLine("Aspect");
            JoinPointContext.Proceed<T>();
            return default(T);
        }

        /*************************************************************************
            */

        /* Test 19.
         * Test of:
         * Use of the opcode "initobj".
         * Return generic type (primitive) - using default without proceed
         * Return generic type (string) - using default without proceed
         * Return generic type (external) - using default without proceed
         * Return generic type (target) - using default without proceed
         * Return generic type (void) - using default without proceed
         */
        public T Test19Aspect<T>()
        {
            Console.WriteLine("Aspect");
            return default(T);
        }

    }

    /*************************************************************************/

    //Used for test 7 as returntype
    public class AspectType
    { }
}
```

Aspect2.cs - Aspect file for interception and introduction tests

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAW.API;

namespace Aspect2
{
    #region Test10

    /* Test 10.
     * Test of:
```

```
 * Class with reference to:                Aspect Assembly
 * Method with reference to:               Aspect Assembly
 * Use of opcode: Accessing local variable (In Test10String).
 * Use of opcode: newobj (In Test10String when creating new Test10InserClass)
 */
public class Test10Class
{
    public int Test10Aspect()
    {
        return Test10Int();
    }

    public string Test10Aspect(int i)
    {
        return Test10String(i);
    }


    public int Test10Int()
    {
        return new Test10InsertClass().IntTest;
    }

    private string Test10String(int i)
    {
        Test10InsertClass t = new Test10InsertClass();
        t.IntTest = i;
        return t.String;
    }
}

public class Test10InsertClass
{
    public static int i = 42;

    public int IntTest
    {
        get { return new NestedClass().ReturnInt(); }
        set { i = value; }
    }

    public string String
    {
        get { return "Aspect" + i; }
    }

    public class NestedClass
    {

        public int ReturnInt()
        {
            return i;
        }
    }
}

#endregion
/*****************************************************************************/

/* Test 11.
 * Test of:
 * Return generic type (void) - with proceed
```

```
 */
public class Test11Class
{
    public T Test11Aspect<T>()
    {
        T t = JoinPointContext.Proceed<T>();
        Test11Method(JoinPointContext.Name);
        return t;
    }

    public void Test11Method(string s)
    {
        Console.WriteLine(s);
    }

}
/*****************************************************************************/

#region Test12

/* Test 12.
 * Test of:
 * Return defined primitive - with proceed
 * Class with reference to:                Mscorlib
 * Method with reference to:               Mscorlib
 * Field with reference to:                Mscorlib
 * Property with reference to:             Mscorlib
 * Event with reference to:                Mscorlib
 * Property with reference to:             Aspect Assembly
 * Use of opcode: unbox (int Test12Return3, when adding object i).
 */
public class Test12InsertClass
{
    int i;

    public Test12InsertClass()
    {
        i = 42;
    }
}

public class Test12Class
{

    internal int Test12Aspect()
    {
        int i = JoinPointContext.Proceed<int>();
        IntField = 2;
        return i + a - Test12Return3() + IntField;
    }
    internal int a = 2;

    public int IntField
    {
        get { return 1; }
        set { a = 2; }
    }

    public int Test12Return3()
    {
        int[] ints = new int[1];
        ints[0] = 1;
```

```
            object i = 1;
            int result = ints[0] + (int)i + IntField;
            return result;
        }

        public event System.EventHandler Test12;
    }

    #endregion
    /*************************************************************************/

    /* Test 13.
     * Test of:
     * Use of opcode: Accessing a field (In t.testField = tn.Test13NestedMethod())
        .
     * Class with reference to:                Target assembly
     * Method with reference to:               Target assembly
     * Field with reference to:                Target assembly
     * Property with reference to:             Target assembly
     * Event with reference to:                Target assembly
     */
    internal class Test13Class
    {
        internal T Test13Aspect<T>() where T : Target.Test13.ITest13
        {
            Test13NestedClass tn = new Test13NestedClass();
            T t = JoinPointContext.Proceed<T>();
            t.testField = tn.Test13NestedMethod();
            return t;
        }

        private static Target.Test13.Test13Target target13;

        internal Test13Class()
        {
            target13 = new Target.Test13.Test13Target();
            target13.Test13Event += new Target.Test13.Test13EventHandler(
                target13_Test13Event);
        }

        EventHandler target13_Test13Event()
        {
            Console.WriteLine("Test13Event happend");
            return null;
        }

        internal Target.Test13.Test13Target Target13
        {
            get { return target13; }
        }

        public class Test13NestedClass
        {
            private Target.Test13.Test13Target t13 = new Target.Test13.
                Test13Target();
            internal int Test13NestedMethod()
            {
                return Target.Test13.Test13Target.a + t13.testField;
            }
        }

        private static event Target.Test13.Test13EventHandler Test13AspectEvent;
```

```
}

/*************************************************************************/

/* Test 15.
 * Test of:
 * Return defined Target - with return
 * Field with reference to:                Aspect assembly
 */
public class Test15Class
{
    public string aspectValue = "Aspect";

    public Target.Test15.Test15Target Test15Aspect()
    {
        Console.WriteLine(aspectValue + "Test15 here");
        return JoinPointContext.Proceed<Target.Test15.Test15Target>();
    }
}

/*************************************************************************/

#region Test16

/* Test 16.
 * Test of:
 * Method with reference to:               External assembly
 * Field with reference to:                External assembly
 * Property with reference to:             External assembly
 * Class with reference to:                External assembly
 * Event with reference to:                External assembly
 */
public class Test16Class
{

    public T Test16Aspect<T>()
    {
        TestEvent += new MyTestLib.Test16EventHandler(Test16ClassTestEvent);
        T t = JoinPointContext.Proceed<T>();
        if (t is MyTestLib.Test16External)
            (t as MyTestLib.Test16External).value = TestMethod().value;
        return t;

    }

    protected EventHandler Test16ClassTestEvent()
    {
        return null;
    }

    protected static MyTestLib.Test16External testField;

    protected static MyTestLib.Test16External TestMethod()
    {
        TestProperty = new MyTestLib.Test16External();
        TestProperty.value = "Aspect";
        return TestProperty;
    }

    protected static MyTestLib.Test16External TestProperty
    {
        get { return testField; }
```

```
            set
            {
                TestEvent();
                testField = value;
            }
        }

        protected static event MyTestLib.Test16EventHandler TestEvent;


        public class Test16NestedClass
        {

            public MyTestLib.Test16External nestedTestField = new MyTestLib.
                Test16External();

            public MyTestLib.Test16External NestedTestMethod()
            {
                NestedTestProperty = new MyTestLib.Test16External();
                return NestedTestProperty;
            }

            public MyTestLib.Test16External NestedTestProperty
            {
                get { return nestedTestField; }
                set
                {
                    NestedTestEvent();
                    nestedTestField = value;
                }
            }

            public event MyTestLib.Test16EventHandler NestedTestEvent;
        }

    }

    #endregion

    /*************************************************************************/

    #region Test20

    /* Test 20.
     * Test of:
     * Use of opcode "newarr" (in Test20targetReturn, and Test20AspectReturn).
     */
    public class Test20Class
    {
        public string Test20Aspect()
        {
            test20Field = Test20AspectReturn();
            test20Field.t20 = Test20targetReturn();
            Test20Event += new EventHandler(Test20ClassTest20Event);
            Test20Event2 += new Target.Test20.Test20EventHandler(
                Test20ClassTest20Event2);
            Test20Event(null, null);
            Test20Event2(null, null);
            return Test20StringProperty;
        }

        EventHandler Test20ClassTest20Event2(object sender, EventArgs args)
```

```
        {
            return null;
        }

        void Test20ClassTest20Event(object sender, EventArgs e)
        {
            return;
        }

        internal event System.EventHandler Test20Event;

        internal event Target.Test20.Test20EventHandler Test20Event2;


        private Test20InsertClass test20Field;

        private Test20InsertClass Test20Property
        {
            get { return test20Field; }
        }

        private string Test20StringProperty
        {
            get { return Test20Property.ToString(); }
        }


        private Target.Test20.Test20Target Test20targetReturn()
        {
            Target.Test20.Test20Target[] target20Arr = new Target.Test20.
                Test20Target[1];
            target20Arr[0] = new Target.Test20.Test20Target();
            target20Arr[0].targetValue = "aspect";
            return target20Arr[0];
        }

        private Test20InsertClass Test20AspectReturn()
        {
            Test20InsertClass[] test20Arr = new Test20InsertClass[1];
            test20Arr[0] = new Test20InsertClass();
            test20Arr[0].value = "aspect";
            return test20Arr[0];
        }
    }

    public class Test20InsertClass
    {
        public string value = "";

        public Target.Test20.Test20Target t20;

        public override string ToString()
        {
            return "Test20 " + value + " " +t20.targetValue;
        }
    }

    #endregion

    /*****************************************************************************/

    /* Test 21.
```

```
 * Test of:
 * Change of basetype.
 */
public class Test21Class
{
    public override string ToString()
    {
        return "Aspect";
    }
}

/*****************************************************************************/


/* Test 22.
 * Test of:
 * Implementation of interface.
 */
public class Test22Class : Test22Interface
{
    public string Test22IMethod()
    {
        return "Aspect";
    }

    public string Test22Aspect()
    {
        if (this is Test22Interface)
        return this.Test22IMethod();

    return null;
    }
}

public interface Test22Interface
{
    string Test22IMethod();
}

/*****************************************************************************/

/* Test 23.
 * Test of:
 * JoinPoint API: Proceed<T>
 * JoinPoint API: GetTarget<T>
 * JoinPoint API: AccessSpecfier
 * JoinPoint API: Arguments
 * JoinPoint API: DeclaringType
 * JoinPoint API: IsStatic
 * JoinPoint API: Name
 * JoinPoint API: ReturnType
 * Use of opcode "switch"
 */
public class Test23Class
{
    public string Test23Aspect(int i)
    {
        switch (i)
        {
            case 0:
                return JoinPointContext.Proceed<string>();
            case 1:
```

```
                    return JoinPointContext.AccessSpecifier;
                case 2:
                    return JoinPointContext.Arguments;
                case 3:
                    return JoinPointContext.DeclaringType;
                case 4:
                    return JoinPointContext.IsStatic.ToString();
                case 5:
                    return JoinPointContext.Name;
                case 6:
                    return JoinPointContext.ReturnType;
                case 7:
                    return JoinPointContext.GetTarget<Target.Test23.Test23Target
                        >().ToString();
                default:
                    return "Aspect";
            }
        }
    }

}
```

Target.cs - Target file for targets without external references

```
///
/// This file contains the targets for all the tests, which does not use external
    libraries.
///

using System;
using System.Collections.Generic;
using System.Text;

namespace Target.Interception
{
    /// <summary>
    /// This class contains the targets for the tests, which only uses
        interception.
    /// </summary>
    public class Target
    {
        public string testValue = "";

        public int Test1()
        {
            return 0;
        }

        /*****************************************************************
            */

        public string Test2(string a)
        {
            return a;
        }

        /*****************************************************************
            */

        public static string Test3(string a)
        {
```

```
            return a;
        }

        /*************************************************************************
            */

        public void Test5A()
        {
            Random r = new Random();
            r.Next();
        }

        public int Test5B()
        {
            Random r = new Random();
            r.Next();
            return 42;
        }

        /*************************************************************************
            */

        internal Target Test8(string a)
        {
            Target t = new Target();
            t.testValue = a;
            return t;
        }

        /*************************************************************************
            */

        protected static void Test9(int i)
        {
            Console.WriteLine("Target" + i);
        }

        protected static void Test9B(string s)
        {
            Console.WriteLine("Target" + s);
        }

        /*************************************************************************
            */

        public override bool Equals(object obj)
        {
            if (obj is Target)
                return (obj as Target).testValue.Equals(testValue);
            else
                return base.Equals(obj);
        }

    }
}

/*****************************************************************************/

namespace Target.Test10
{
    public class Test10Target
    {
```

```
        public int Test10A()
        {
            return 0;
        }

        public string Test10B(int i)
        {
            return "" + i;
        }
    }
}

/*****************************************************************************/

namespace Target.Test11
{
    public class Test11Target
    {

        public int Test11()
        {
            return 42;
        }
    }
}

/*****************************************************************************/

namespace Target.Test12
{
    public class Test12Target
    {
        public int Test12(int i)
        {
            return i;
        }
    }
}

/*****************************************************************************/

namespace Target.Test13
{
    public interface ITest13
    {
        int testField
        {
            get;
            set;
        }

    }

    public class Test13Target : ITest13
    {
        public static int a = 2;

        public int testField
        {
            get { return a; }
            set
```

```
            {
                if (Test13Event != null)
                    Test13Event();
                a = value;
            }
        }

        public ITest13 ReturnITest13(ITest13 t)
        {
            return t;
        }

        public int Test13Method()
        {
            return 1;
        }

        public event Test13EventHandler Test13Event;

        public override bool Equals(object obj)
        {
            if (obj is ITest13)
                return (obj as ITest13).testField == testField;

            return base.Equals(obj);
        }

    }

    public delegate System.EventHandler Test13EventHandler();
}

/*****************************************************************************/

namespace Target.Test14
{
    public class Test14Target
    {

        public int Test14()
        {
            return 1;
        }
    }
}

/*****************************************************************************/

namespace Target.Test15
{
    public interface Test15I { }

    public class Test15Target : Test15I
    {
        public string value = "";

        public Test15Target Test15(string a)
        {
            value = a;
            return this;
        }
```

```
        public override bool Equals(object obj)
        {
            if (obj is Test15Target)
                return (obj as Test15Target).value.Equals(value);

            return base.Equals(obj);
        }
    }
}

/*************************************************************************/

namespace Target.Test20
{
    public class Test20Target
    {
        public string targetValue = "target";

        public string Test20()
        {
            return "target";
        }

    }

    public delegate System.EventHandler Test20EventHandler(object sender,
        EventArgs args);
}

/*************************************************************************/

namespace Target.Test21.ns //special namespace, to be target able by xxx.*
{
    public class Test21Target
    {
        public string Test21Method()
        {
            return base.ToString();
        }
    }

}

/*************************************************************************/

namespace Target.Test22
{
    public class Test22Target
    {

        public string Test22Method()
        {
            return "Target";
        }

    }
}

/*************************************************************************/

namespace Target.Test23
{
```

```
    public class Test23Target
    {
        public string Test23Method(int i)
        {
            return i.ToString();
        }

        public override string ToString()
        {
            return "TargetToString";
        }
    }
}
/*****************************************************************************/
```

Target2.cs - Target file for targets with external references

```
///
/// This file contains the targets for all the tests, which does use external
    libraries.
///


using System;
using System.Collections.Generic;
using System.Text;

/*****************************************************************************/

namespace Target.Test4
{
    /// <summary>
    ///
    /// </summary>
    public class Test4Target
    {
        public MyTestLib.MyType Test4A()
        {
            return new MyTestLib.MyType();
        }

        public MyTestLib.MyType Test4B(string a)
        {
            MyTestLib.MyType returnvalue = new MyTestLib.MyType();
            returnvalue.Name = a;
            return returnvalue;
        }

        public MyTestLib.MyType Test4C(string a, string b)
        {
            MyTestLib.MyType returnvalue = new MyTestLib.MyType();
            returnvalue.Name = a + " " + b;
            return returnvalue;
        }

    }
}

/*****************************************************************************/

namespace Target.Test6
```

```
{
    public class Test6Target
    {
        public MyTestLib.MyType Test6(MyTestLib.MyType m)
        {
            return m;
        }

    }
}

/*************************************************************************/

namespace Target.Test16
{
    public class Test16Target
    {
        public MyTestLib.Test16External Test16()
        {
            return new MyTestLib.Test16External();
        }
    }
}

/*************************************************************************/

namespace Target.Test17
{
    public class Test17Target
    {
        public int Test17(int i)
        {
            return i;
        }

        public string Test17(string s)
        {
            return s;
        }

        public MyTestLib.Test16External Test17(MyTestLib.Test16External t)
        {
            return t;
        }

        public Test17Target Test17(Test17Target t)
        {
            return t;
        }
    }
}

/*************************************************************************/

namespace Target.Test18
{
    public class Test18Target
    {
        public void Test18()
        {
        }
```

```
        public int Test18int()
        {
            return 42;
        }

        public string Test18string()
        {
            return "42";
        }

        public Test18Target Test18target()
        {
            return new Test18Target();
        }

        public MyTestLib.Test16External Test18external()
        {
            return new MyTestLib.Test16External();
        }
    }
}
```

Target3.cs - Target file for targets with references to aspect

```
///
/// This file contains the targets for all the tests, which uses references to the
    Aspect assemblies.
///
using System;
using System.Collections.Generic;
using System.Text;

namespace Target.Test7
{
    public class Test7Target
    {
        private Aspect.AspectType Test7()
        {
            return null;
        }
    }
}
```

MyType.cs - File used as external reference

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MyTestLib
{
    public class MyType
    {
        private string _name = "MyType - Name";
        public string Name
        {
            get { return _name; }
            set { _name = value; }

        }
```

```
        public override bool Equals(object obj)
        {
            if (obj is MyType)
                return (obj as MyType).Name.Equals(_name);

            return base.Equals(obj);
        }
    }

    public class Test16External
    {
        public string value = "";

        public override bool Equals(object obj)
        {
            if (obj is Test16External)
                return (obj as Test16External).value.Equals(value);

            return base.Equals(obj);
        }
    }

    public delegate System.EventHandler Test16EventHandler();
}
```

ErrorLogger.cs - The logger

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAWTester
{
    /// <summary>
    /// Used to store error messages and print them out when needed.
    /// </summary>
    public class ErrorLogger
    {
        private StringBuilder _sbError;
        private StringBuilder _sbWarning;
        private int _ErrorCounter = 0;
        private int _WarningCounter = 0;

        public ErrorLogger()
        {
            _sbError = new StringBuilder();
            _sbWarning = new StringBuilder();
        }

        /// <summary>
        /// Add an error to the log.
        /// </summary>
        /// <param name="error">The error to add to the log.</param>
        public void AddError(string error)
        {
            _ErrorCounter++;
            _sbError.Append("\nERROR: " +error+"\n");
        }

        /// <summary>
        /// Prints the full log out on the console.out.
```

```
        /// </summary>
        /// <param name="start">A string that will be inserted at the start of the
             printout.</param>
        internal void Print(string start)
        {
            Console.WriteLine("\n\n----------------------");
            Console.WriteLine(start+"\n");
            Console.WriteLine("Testing generated {0} warning(s) and {1} error(s)",
                _WarningCounter,_ErrorCounter);
            Console.WriteLine(_sbWarning.ToString());
            Console.WriteLine(_sbError.ToString());
        }

        /// <summary>
        /// Add a warning to the log.
        /// </summary>
        /// <param name="warning">The warning to add to the log.</param>
        internal void AddWarning(string warning)
        {
            _WarningCounter++;
            _sbWarning.Append("WARNING: " + warning + "\n");

        }
    }
}
```

Appendix S

Source code for YIIHAW - API

JoinPointContext.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.API
{
    /// <summary>
    /// The join point API. Only to be used in advice methods.
    /// </summary>
    public class JoinPointContext
    {
        /// <summary>
        /// Invokes the original target method
        /// </summary>
        /// <typeparam name="T">The return type of the advice method</typeparam>
        /// <returns>A value of the specified type</returns>
        public static T Proceed<T>()
        {
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                method can only be invoked from an advice method.");
        }

        /// <summary>
        /// Returns a reference to object being intercepted (instance methods only
            )
        /// </summary>
        /// <typeparam name="T">The declaring type of the target method</typeparam
            >
        /// <returns>An reference to the object being intercepted</returns>
        public static T GetTarget<T>()
        {
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                method can only be accessed from an advice method.");
        }

        /// <summary>
        /// Returns a string describing the declaring type (namespace and class)
            of the target method
        /// </summary>
        public static string DeclaringType
        {
            get
            {
```

```
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                property can only be accessed from an advice method.");
        }
    }

    /// <summary>
    /// Returns a string describing the name of the target method
    /// </summary>
    public static string Name
    {
        get
        {
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                property can only be accessed from an advice method.");
        }
    }

    /// <summary>
    /// Returns a string describing the return type of the target method
    /// </summary>
    public static string ReturnType
    {
        get
        {
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                property can only be accessed from an advice method.");
        }
    }

    /// <summary>
    /// Returns a string describing the access specifier of the target method
    /// </summary>
    public static string AccessSpecifier
    {
        get
        {
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                property can only be accessed from an advice method.");
        }
    }

    /// <summary>
    /// Returns a boolean indicating if the target method is static or not
    /// </summary>
    public static bool IsStatic
    {
        get
        {
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                property can only be accessed from an advice method.");
        }
    }

    /// <summary>
    /// Returns a commma-seperated list of arguments of the target method
    /// </summary>
    public static string Arguments
    {
        get
        {
            throw new YIIHAW.Exceptions.NotSupportedOperationException("This
                property can only be accessed from an advice method.");
```

```
            }
        }

    }
}
```

Void.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.API
{
    /// <summary>
    /// The join point API. Use when you need to invoke Proceed() with type "void
        ".
    /// </summary>
    public class Void
    {
        private Void() { } // no instance allowed
    }
}
```

Appendix T

Source code for YIIHAW - Exceptions

Exceptions.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.Exceptions
{
    /// <summary>
    /// Exception thrown when an operation is not supported.
    /// </summary>
    public class NotSupportedOperationException : System.Exception
    {
        public NotSupportedOperationException(string message) : base(message)
        {
        }
    }

    /// <summary>
    /// Exception thrown when an illegal operation is met.
    /// </summary>
    public class IllegalOperationException : System.Exception
    {
        public IllegalOperationException(string message)
            : base(message)
        {
        }
    }

    /// <summary>
    /// Exception thrown when a construct could not be found.
    /// </summary>
    public class ConstructNotFoundException : System.Exception
    {
        public ConstructNotFoundException(string message)
            : base(message)
        {
        }
    }

    /// <summary>
    /// Exception thrown when an internal error in Yiihaw has happend.
```

```
    /// </summary>
    public class InternalErrorException : System.Exception
    {
        public InternalErrorException(string message)
            : base(message)
        {
        }
    }
}
```

Appendix U

Source code for YIIHAW - Output

OutputFormatter.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.Output
{
    /// <summary>
    /// Used for all formatting and storing the log output from the weaver.
    /// </summary>
    public static class OutputFormatter
    {
        private static LinkedList<string> _warnings = new LinkedList<string>();
        private static string _exception = null;
        private static string _internalException = null;
        private static LinkedList<string> _methodsIntercepted = new LinkedList<
            string>();
        private static LinkedList<string> _methodsNotIntercepted = new LinkedList<
            string>();
        private static Dictionary<string, List<string>> _methodsIntroduced = new
            Dictionary<string, List<string>>();
        private static Dictionary<string, List<string>> _propertiesIntroduced =
            new Dictionary<string, List<string>>();
        private static Dictionary<string, List<string>> _fieldsIntroduced = new
            Dictionary<string, List<string>>();
        private static Dictionary<string, List<string>> _typesIntroduced = new
            Dictionary<string, List<string>>();
        private static Dictionary<string, List<string>> _eventsIntroduced = new
            Dictionary<string, List<string>>();

        /// <summary>
        /// Resets all information stored in the log.
        /// </summary>
        public static void Reset()
        {
        _warnings = new LinkedList<string>();
        _exception = null;
        _internalException = null;
        _methodsIntercepted = new LinkedList<string>();
        _methodsNotIntercepted = new LinkedList<string>();
        _methodsIntroduced = new Dictionary<string, List<string>>();
        _propertiesIntroduced = new Dictionary<string, List<string>>();
        _fieldsIntroduced = new Dictionary<string, List<string>>();
        _typesIntroduced = new Dictionary<string, List<string>>();
```

```
_eventsIntroduced = new Dictionary<string, List<string>>();
}

/// <summary>
/// Add a warning to the log.
/// </summary>
/// <param name="message">The message to store as warning.</param>
public static void AddWarning(string message)
{
    _warnings.AddLast(message);
}

/// <summary>
/// Add an exception to the log.
/// </summary>
/// <param name="message">The message to store as the exception.</param>
public static void AddException(string message)
{
    _exception = message;
}

/// <summary>
/// Add an internal exception to the log.
/// Only one exception stored here, as it is expected that the program
    will stop when an internal exception is added.
/// </summary>
/// <param name="message">The message that will be stored as the internal
    exception.</param>
public static void AddInternalException(string message)
{
    _internalException = message;
}

/// <summary>
/// Add information about a method that has been intercepted by a given
    advice.
/// </summary>
/// <param name="method">The method that has been intercepted.</param>
/// <param name="advice">The advice which was used in the interception.</
    param>
public static void AddMethodIntercepted(string method, string advice)
{
    _methodsIntercepted.AddLast(method + " using " + advice);
}

/// <summary>
/// Add information about a method that matched the target part of the
    pointcut,
/// but where there was no suitable advice to use for intercepting the
    method.
/// </summary>
/// <param name="method">The method that was not intercepted.</param>
public static void AddMethodNotIntercepted(string method)
{
    _methodsNotIntercepted.AddLast(method);
}

/// <summary>
/// Add information about a method that has been inserted into a target
    type.
/// </summary>
/// <param name="aspect">The method that has been inserted.</param>
```

```
/// <param name="target">The type that the method has been inserted into
    .</param>
public static void AddMethod(string aspect, string target)
{
    if (!_methodsIntroduced.ContainsKey(aspect))
        _methodsIntroduced[aspect] = new List<string>();

    _methodsIntroduced[aspect].Add(target);
}

/// <summary>
/// Add information about a property that has been inserted into a target
    type.
/// </summary>
/// <param name="aspect">The property that has been inserted.</param>
/// <param name="target">The type that the property has been inserted into
    .</param>
public static void AddProperty(string aspect, string target)
{
    if (!_propertiesIntroduced.ContainsKey(aspect))
        _propertiesIntroduced[aspect] = new List<string>();

    _propertiesIntroduced[aspect].Add(target);
}

/// <summary>
/// Add information about a field that has been inserted into a target
    type.
/// </summary>
/// <param name="aspect">The field that has been inserted.</param>
/// <param name="target">The type that the field has been inserted into.</
    param>
public static void AddField(string aspect, string target)
{
    if (!_fieldsIntroduced.ContainsKey(aspect))
        _fieldsIntroduced[aspect] = new List<string>();

    _fieldsIntroduced[aspect].Add(target);
}

/// <summary>
/// Add information about an event that has been inserted into a target
    type.
/// </summary>
/// <param name="aspect">The event that has been inserted.</param>
/// <param name="target">The type that the event has been inserted into.</
    param>
public static void AddEvent(string aspect, string target)
{
    if (!_eventsIntroduced.ContainsKey(aspect))
        _eventsIntroduced[aspect] = new List<string>();

    _eventsIntroduced[aspect].Add(target);
}

/// <summary>
/// Add information about a type that has been inserted into a target type
     or namespace.
/// </summary>
/// <param name="aspect">The type that has been inserted.</param>
/// <param name="target">The type or namespace that the type has been
    inserted into.</param>
```

```
public static void AddType(string aspect, string target)
{
    if (!_typesIntroduced.ContainsKey(aspect))
        _typesIntroduced[aspect] = new List<string>();

    _typesIntroduced[aspect].Add(target);
}

/// <summary>
/// Prints the logged output on the console.out.
/// </summary>
/// <param name="verbose">A boolean indicating if the output should be
    verbose (include detailed information about the weaving).</param>
public static void PrintOutput(bool verbose)
{
    string splitter = "-----------------------------------------\n";
    StringBuilder sb = new StringBuilder();

    // check if there was exceptions logged.
    if (_exception != null)
    {
        sb.Append("Fatal error: ");
        sb.Append(_exception);
    }
    else if (_internalException != null)
    {
        sb.Append("Internal error: ");
        sb.Append(_internalException);
    }
    else
    {   // print warnings.
        foreach (string warning in _warnings)
        {
            sb.Append("WARNING: ");
            sb.Append(warning);
            sb.Append("\n");
        }
        if (_warnings.Count > 0)
            sb.Append("\n");

        if (verbose) // verbose feature is selected - print detailed
            information about the constructs weaved
        {
            // print methods intercepted.
            if (_methodsIntercepted.Count > 0)
            {
                sb.Append("The following methods were succesfully
                    intercepted: \n");

                foreach (string methodIntercepted in _methodsIntercepted)
                {
                    sb.Append("- ");
                    sb.Append(methodIntercepted);
                    sb.Append("\n");
                }
            }
            else
                sb.Append("\nNo methods were intercepted.\n");

            // print methods not intercepted.
            if (_methodsNotIntercepted.Count > 0)
            {
```

```
        sb.Append("\nNo suitable advice were found for the
            following methods:\n");
        foreach (string methodNotIntercepted in
            _methodsNotIntercepted)
        {
            sb.Append("- ");
            sb.Append(methodNotIntercepted);
            sb.Append("\n");
        }
    }

    //print methods introduced.
    if (_methodsIntroduced.Count > 0)
    {
        sb.Append("\nThe following methods were introduced:\n");
        foreach (string aspect in _methodsIntroduced.Keys)
        {
            sb.Append("- ");
            sb.Append(aspect);
            sb.Append("\n");

            foreach (string target in _methodsIntroduced[aspect])
            {
                sb.Append("\t at ");
                sb.Append(target);
                sb.Append("\n");
            }
        }
    }
    else
        sb.Append("\nNo methods were introduced.\n");

    // print properties introduced.
    if (_propertiesIntroduced.Count > 0)
    {
        sb.Append("\nThe following properties were introduced:\n")
            ;
        foreach (string aspect in _propertiesIntroduced.Keys)
        {
            sb.Append("- ");
            sb.Append(aspect);
            sb.Append("\n");

            foreach (string target in _propertiesIntroduced[aspect
                ])
            {
                sb.Append("\t at ");
                sb.Append(target);
                sb.Append("\n");
            }
        }
    }
    else
        sb.Append("\nNo properties were introduced.\n");

    // print fields introduced.
    if (_fieldsIntroduced.Count > 0)
    {
        sb.Append("\nThe following fields were introduced:\n");
        foreach (string aspect in _fieldsIntroduced.Keys)
        {
            sb.Append("- ");
```

```
                sb.Append(aspect);
                sb.Append("\n");

                foreach (string target in _fieldsIntroduced[aspect])
                {
                    sb.Append("\t at ");
                    sb.Append(target);
                    sb.Append("\n");
                }
            }
        }
        else
            sb.Append("\nNo fields were introduced.\n");

        // print events introduced.
        if (_eventsIntroduced.Count > 0)
        {
            sb.Append("\nThe following events were introduced:\n");
            foreach (string aspect in _eventsIntroduced.Keys)
            {
                sb.Append("- ");
                sb.Append(aspect);
                sb.Append("\n");

                foreach (string target in _eventsIntroduced[aspect])
                {
                    sb.Append("\t at ");
                    sb.Append(target);
                    sb.Append("\n");
                }
            }
        }
        else
            sb.Append("\nNo events were introduced.\n");

        // print types introduced.
        if (_typesIntroduced.Count > 0)
        {
            sb.Append("\nThe following classes and interfaces were
                introduced:\n");
            foreach (string aspect in _typesIntroduced.Keys)
            {
                sb.Append("- ");
                sb.Append(aspect);
                sb.Append("\n");

                foreach (string target in _typesIntroduced[aspect])
                {
                    sb.Append("\t at ");
                    sb.Append(target);
                    sb.Append("\n");
                }
            }
        }
        else
            sb.Append("\nNo classes or interfaces were introduced.\n")
                ;


        sb.Append("\n" + splitter);
    }
```

```
            // print the total number of methods that were intercepted
            sb.Append("Methods intercepted: ");
            sb.Append(_methodsIntercepted.Count);
            sb.Append("\n");

            // print number of methods that was targeted, but was not
                intercepted
            sb.Append("Methods targeted, but not intercepted: ");
            sb.Append(_methodsNotIntercepted.Count);
            sb.Append("\n");

            // print number of methods introduced
            sb.Append(_methodsIntroduced.Count);
            sb.Append(" method(s) were introduced at ");
            sb.Append(TotalIntroductions(_methodsIntroduced));
            sb.Append(" location(s).\n");

            // print number of properties introduced
            sb.Append(_propertiesIntroduced.Count);
            sb.Append(" properties were introduced at ");
            sb.Append(TotalIntroductions(_propertiesIntroduced));
            sb.Append(" location(s).\n");

            // print number of fields introduced
            sb.Append(_fieldsIntroduced.Count);
            sb.Append(" field(s) were introduced at ");
            sb.Append(TotalIntroductions(_fieldsIntroduced));
            sb.Append(" location(s).\n");

            // print number of events introduced
            sb.Append(_eventsIntroduced.Count);
            sb.Append(" event(s) were introduced at ");
            sb.Append(TotalIntroductions(_eventsIntroduced));
            sb.Append(" location(s).\n");

            // print number of classes and interfaces introduced
            sb.Append(_typesIntroduced.Count);
            sb.Append(" class(es) or interface(s) were introduced at ");
            sb.Append(TotalIntroductions(_typesIntroduced));
            sb.Append(" location(s).\n");

            // print number of errors and warnings
            sb.Append(splitter);
            sb.Append(_warnings.Count);
            sb.Append(" warning(s)");
        }
        Console.WriteLine(sb.ToString());
    }

    /// <summary>
    /// Calculate the total number of a certain type of introductions.
    /// </summary>
    /// <param name="dictionary">A dictionary containing the log information
        about the introduction type.</param>
    /// <returns>The number of times the type of introduction has happend.</
        returns>
    private static int TotalIntroductions(Dictionary<string, List<string>>
        dictionary)
    {
        int total = 0;
        foreach (List<string> list in dictionary.Values)
            total += list.Count;
```

```
            return total;
        }

        /// <summary>
        /// Given a name of a type, the types short form is returned.
        /// There will only be a change between the parameter and the return value
            , when the type
        /// is one of the primitive types in .NET, and it is specified in its long
             form e.g. System.Int32
        /// </summary>
        /// <param name="type">The name of the type.</param>
        /// <returns>The possible shorter form of the type.</returns>
        public static string GetTypeShortFormat(string type)
        {
            switch (type)
            {
                case "System.Byte":
                    return "byte";
                case "System.SByte":
                    return "sbyte";
                case "System.Int32":
                    return "int";
                case "System.UInt32":
                    return "uint";
                case "System.Int16":
                    return "short";
                case "System.UInt16":
                    return "ushort";
                case "System.Int64":
                    return "long";
                case "System.UInt64":
                    return "ulong";
                case "System.Single":
                    return "float";
                case "System.Double":
                    return "double";
                case "System.Char":
                    return "char";
                case "System.Boolean":
                    return "bool";
                case "System.Object":
                    return "object";
                case "System.String":
                    return "string";
                case "System.Decimal":
                    return "decimal";
                case "System.Void":
                    return "void";
                default:
                    return type;
            }
        }
    }
}
```

Appendix V

Source code for YIIHAW - Pointcut

Tokenizer.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.Pointcut.LexicalAnalysis
{
    /// <summary>
    /// Splits a string into lexical tokens.
    /// </summary>
    public class Tokenizer
    {
        /// <summary>
        /// Determines the type of token just encountered. Possible values are EOF
            (End Of File), WORD (string literal) or NUMBER (integer).
        /// </summary>
        public enum TokenType { EOF, WORD, NUMBER };

        private string _str;
        private int _index = 0;
        private TokenType _tokentype = TokenType.EOF;
        private string _sval = "";
        private int _nval = 0;
        private LinkedList<char> _seperators;

        /// <summary>
        /// Creates an instance of the Tokenizer class.
        /// </summary>
        /// <param name="str">The string to tokenize.</param>
        public Tokenizer(string str)
        {
            _seperators = new LinkedList<char>();
            _str = str;
            Next(); // fetch the first token
        }

        /// <summary>
        /// When a WORD token is encountered, this property contains the string
            value of the token.
        /// </summary>
        public string Sval
        {
            get
            {
```

```
            return _sval;
        }
    }

    /// <summary>
    /// When a NUMBER token is encountered, this property contains the integer
         value of the token.
    /// </summary>
    public int Nval
    {
        get
        {
            return _nval;
        }
    }

    /// <summary>
    /// The type of token just encountered.
    /// </summary>
    public TokenType Token
    {
        get
        {
            return _tokentype;
        }
    }

    /// <summary>
    /// Adds a seperator value. Seperator values will be extracted from the
        input string and returned as a token even if these seperators are not
        enclosed by whitespace characters.
    /// </summary>
    /// <param name="seperator">The seperator value to extract from the input
        string.</param>
    public void AddSeperator(char seperator)
    {
        _seperators.AddLast(seperator);
    }

    /// <summary>
    /// Fetches the next token from the input string.
    /// </summary>
    public void Next()
    {
        if ((_index + 1) > _str.Length)
            _tokentype = TokenType.EOF;
        else
        {
            // skip ahead to the first non-whitespace character
            while (_index < _str.Length && _str[_index].Equals(' '))
                _index++;

            // first character found - find the next token
            int startindex = _index;
            while (_index < _str.Length && !(_str[_index].Equals(' ') ||
                IsASeperator(_str[_index])))
                _index++;

            if (startindex == _index) // index variable has not been
                incremented - this means that the current token is a seperator
                - increment the index variable by 1 and continue
                _index++;
```

```
                // fetch token
                string token = _str.Substring(startindex, _index - startindex);

                // token has been found - check the token type
                if (int.TryParse(token, out _nval)) // token is a number
                    _tokentype = TokenType.NUMBER;
                else // token is a word
                {
                    _tokentype = TokenType.WORD;
                    _sval = token;
                }

            }
        }

        /// <summary>
        /// Determines if a given character in the input string is a seperator.
        /// </summary>
        /// <param name="c">The character to check.</param>
        /// <returns>A boolean indicating whether the character is a seperator or
            not.</returns>
        private bool IsASeperator(char c)
        {
            foreach (char seperator in _seperators)
                if (c.Equals(seperator))
                    return true;

            return false;
        }
    }
}
```

Scanner.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace YIIHAW.Pointcut.LexicalAnalysis
{
    /// <summary>
    /// Scans the contents of a file and identifies keywords.
    /// </summary>
    public class Scanner
    {
        /// <summary>
        /// Determines the type of token just encountered.
        /// </summary>
        public enum TokenType { EOF, EOS, AROUND, INSERT, MODIFY, PUBLIC, PRIVATE,
            PROTECTED, INTERNAL, STATIC, INSTANCE, ANY, VOID, NAME, COMMA, COLON,
            PAROPEN, PARCLOSE, INHERITS, DO, CLASS, INTERFACE, METHOD, PROPERTY,
           DELEGATE, FIELD, INTO, IMPLEMENT, EVENT, ENUM, ATTRIBUTE };

        private Tokenizer _tokenizer;
        private TokenType _tokentype;
        private string _sval = "";

        /// <summary>
        /// Creates an instance of the Scanner class.
```

```
/// </summary>
/// <param name="filename">The name of the input file.</param>
public Scanner(string filename)
{
    string input = "";

    try
    {
        using (StreamReader reader = new StreamReader(filename))
            while (!reader.EndOfStream)
            {
                string line = reader.ReadLine();
                if (!line.StartsWith("//"))
                    input += line.Trim();
            }
    }
    catch (IOException)
    {
        throw new YIIHAW.Pointcut.Exceptions.InputFileNotFoundException("
            Error opening file: " + filename);
    }

    _tokenizer = new Tokenizer(input);
    _tokenizer.AddSeperator(':');
    _tokenizer.AddSeperator(';');
    _tokenizer.AddSeperator('(');
    _tokenizer.AddSeperator(')');
    _tokenizer.AddSeperator(',');

    DetermineTokenType();
}

/// <summary>
/// When a NAME token is encountered, this property contains the string
    value of the token.
/// </summary>
public string Sval
{
    get
    {
        return _sval;
    }
}

/// <summary>
/// The type of token just encountered.
/// </summary>
public TokenType Token
{
    get
    {
        return _tokentype;
    }
}

/// <summary>
/// Fetches the next token.
/// </summary>
public void Next()
{
    if (_tokenizer.Token != Tokenizer.TokenType.EOF) // end-of-file has
        not been reached yet
```

```
        {
            _tokenizer.Next(); // fetch next token

            DetermineTokenType();
        }
    }

    /// <summary>
    /// Reads the last token that was fetched and determines its type.
    /// </summary>
    private void DetermineTokenType()
    {
        if (_tokenizer.Token == Tokenizer.TokenType.EOF)
            _tokentype = TokenType.EOF;
        else if (_tokenizer.Token == Tokenizer.TokenType.WORD)
        {
            switch (_tokenizer.Sval)
            {
                case ";":
                    _tokentype = TokenType.EOS;
                    break;
                case "around":
                    _tokentype = TokenType.AROUND;
                    break;
                case "insert":
                    _tokentype = TokenType.INSERT;
                    break;
                case "modify":
                    _tokentype = TokenType.MODIFY;
                    break;
                case "public":
                    _tokentype = TokenType.PUBLIC;
                    break;
                case "private":
                    _tokentype = TokenType.PRIVATE;
                    break;
                case "protected":
                    _tokentype = TokenType.PROTECTED;
                    break;
                case "internal":
                    _tokentype = TokenType.INTERNAL;
                    break;
                case "static":
                    _tokentype = TokenType.STATIC;
                    break;
                case "instance":
                    _tokentype = TokenType.INSTANCE;
                    break;
                case "*":
                    _tokentype = TokenType.ANY;
                    break;
                case "void":
                    _tokentype = TokenType.VOID;
                    break;
                case ",":
                    _tokentype = TokenType.COMMA;
                    break;
                case ":":
                    _tokentype = TokenType.COLON;
                    break;
                case "(":
                    _tokentype = TokenType.PAROPEN;
```

```
                        break;
                    case ")":
                        _tokentype = TokenType.PARCLOSE;
                        break;
                    case "inherits":
                    case "inherit":
                        _tokentype = TokenType.INHERITS;
                        break;
                    case "do":
                        _tokentype = TokenType.DO;
                        break;
                    case "class":
                        _tokentype = TokenType.CLASS;
                        break;
                    case "interface":
                        _tokentype = TokenType.INTERFACE;
                        break;
                    case "method":
                        _tokentype = TokenType.METHOD;
                        break;
                    case "property":
                        _tokentype = TokenType.PROPERTY;
                        break;
                    case "delegate":
                        _tokentype = TokenType.DELEGATE;
                        break;
                    case "field":
                        _tokentype = TokenType.FIELD;
                        break;
                    case "into":
                        _tokentype = TokenType.INTO;
                        break;
                    case "implement":
                    case "implements":
                        _tokentype = TokenType.IMPLEMENT;
                        break;
                    case "event":
                        _tokentype = TokenType.EVENT;
                        break;
                    case "enum":
                    case "enumeration":
                        _tokentype = TokenType.ENUM;
                        break;
                    case "attribute":
                        _tokentype = TokenType.ATTRIBUTE;
                        break;
                    default:
                        _tokentype = TokenType.NAME;
                        _sval = _tokenizer.Sval;
                        break;
                }
            }
        }
    }
}
```

Parser.cs

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace YIIHAW.Pointcut
{
    /// <summary>
    /// Determines the access specifier.
    /// </summary>
    public enum AccessEnum { PUBLIC, PRIVATE, PROTECTED, INTERNAL, ANY,
        NOTAVAILABLE };

    /// <summary>
    /// Determines the member type (static, instance, *)
    /// </summary>
    public enum InvocationKindEnum { STATIC, INSTANCE, ANY, NOTAVAILABLE };

    /// <summary>
    /// Determines the type of return value (void, *, specific).
    /// </summary>
    public enum ReturnTypeEnum { VOID, ANY, SPECIFIC, NOTAVAILABLE };

    /// <summary>
    /// Determines the type of argument (none, *, specific).
    /// </summary>
    public enum ArgTypeEnum { NONE, ANY, SPECIFIC };

    /// <summary>
    /// Determines the type of inheritance requirement (*, specific).
    /// </summary>
    public enum InheritTypeEnum { ANY, SPECIFIC };

    /// <summary>
    /// Determines the type of insert statement.
    /// </summary>
    public enum InsertTypeEnum { CLASS, METHOD, DELEGATE, FIELD, PROPERTY, EVENT,
        ENUM, ATTRIBUTE };

    /// <summary>
    /// Determines the type of modify statement
    /// </summary>
    public enum ModifyTypeEnum { INHERIT, IMPLEMENT };

    /// <summary>
    /// Stores information for an around pointcut.
    /// </summary>
    public struct Around
    {
        private AccessEnum _access;
        private InvocationKindEnum _invocationKind;
        private ReturnType _returnType;
        private string _targetType;
        private Method _targetMethod;
        private Inherit _inherit;
        private string _adviceType;
        private string _adviceName;

        /// <summary>
        /// Creates an instance of the Around struct.
        /// </summary>
        public Around(AccessEnum access, InvocationKindEnum invocationKind,
            ReturnType returnType, string targetType, Method targetMethod, Inherit
             inherit, string adviceType, string adviceName)
        {
            _access = access;
```

```
        _invocationKind = invocationKind;
        _returnType = returnType;
        _targetType = targetType;
        _targetMethod = targetMethod;
        _inherit = inherit;
        _adviceType = adviceType;
        _adviceName = adviceName;
    }

    /// <summary>
    /// The access specifier.
    /// </summary>
    public AccessEnum Access
    {
        get
        {
            return _access;
        }
    }

    /// <summary>
    /// The member type (instance, static, any).
    /// </summary>
    public InvocationKindEnum InvocationrType
    {
        get
        {
            return _invocationKind;
        }
    }

    /// <summary>
    /// The return type of the target type.
    /// </summary>
    public ReturnType ReturnType
    {
        get
        {
            return _returnType;
        }
    }

    /// <summary>
    /// The target type (namespace and class).
    /// </summary>
    public string TargetType
    {
        get
        {
            return _targetType;
        }
    }

    /// <summary>
    /// The target method.
    /// </summary>
    public Method TargetMethod
    {
        get
        {
            return _targetMethod;
        }
```

```
        }

        /// <summary>
        /// The inheritance constraint.
        /// </summary>
        public Inherit Inherit
        {
            get
            {
                return _inherit;
            }
        }

        /// <summary>
        /// The advice type (namespace and class).
        /// </summary>
        public string AdviceType
        {
            get
            {
                return _adviceType;
            }
        }

        /// <summary>
        /// The advice method.
        /// </summary>
        public string AdviceName
        {
            get
            {
                return _adviceName;
            }
        }

        /// <summary>
        /// A textual representation of the around pointcut.
        /// </summary>
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append("around ");
            sb.Append(_access.ToString());
            sb.Append(" ");
            sb.Append(_invocationKind.ToString());
            sb.Append(" ");
            sb.Append(_returnType.ToString());
            sb.Append(" ");
            sb.Append(_targetType);
            sb.Append(":");
            sb.Append(_targetMethod.ToString());
            sb.Append(" ");
            sb.Append(_inherit.ToString());
            sb.Append(" do ");
            sb.Append(_adviceType);
            sb.Append(":");
            sb.Append(_adviceName);
            sb.Append(";");
            return sb.ToString();
        }
    }
```

```
/// <summary>
/// Stores information for an insert pointcut.
/// </summary>
public struct Insert
{
    private InsertTypeEnum _insertType;
    private AccessEnum _access;
    private InvocationKindEnum _invocationKind;
    private ReturnType _returnType;
    private string _type;
    private string _name;
    private Method _method;
    private string _targetType;

    /// <summary>
    /// Creates an instance of the Insert struct.
    /// </summary>
    public Insert(InsertTypeEnum insertType, AccessEnum access,
        InvocationKindEnum invocationKind, ReturnType returnType, string type,
         string name, string targetType)
    {
        _insertType = insertType;
        _access = access;
        _invocationKind = invocationKind;
        _returnType = returnType;
        _type = type;
        _name = name;
        _method = new Method(); // default value
        _targetType = targetType;
    }

    /// <summary>
    /// Creates an instance of the Insert struct.
    /// </summary>
    public Insert(InsertTypeEnum insertType, AccessEnum access,
        InvocationKindEnum invocationKind, ReturnType returnType, string type,
         Method method, string targetType)
    {
        _insertType = insertType;
        _access = access;
        _invocationKind = invocationKind;
        _returnType = returnType;
        _type = type;
        _name = ""; // default value
        _method = method;
        _targetType = targetType;
    }

    /// <summary>
    /// Creates an instance of the Insert struct.
    /// </summary>
    public Insert(InsertTypeEnum insertType, string type, string targetType)
    {
        _insertType = insertType;
        _access = AccessEnum.NOTAVAILABLE; // default value
        _invocationKind = InvocationKindEnum.NOTAVAILABLE; // default value
        _returnType = new ReturnType(ReturnTypeEnum.NOTAVAILABLE, ""); //
            default value
        _type = type;
        _name = ""; // default value
        _method = new Method(); // default value
        _targetType = targetType;
```

```
}

/// <summary>
/// The type of introduction.
/// </summary>
public InsertTypeEnum InsertType
{
    get
    {
        return _insertType;
    }
}

/// <summary>
/// The access specifier.
/// </summary>
public AccessEnum Access
{
    get
    {
        return _access;
    }
}

/// <summary>
/// The member type (instance, static, any).
/// </summary>
public InvocationKindEnum InvocationKind
{
    get
    {
        return _invocationKind;
    }
}

/// <summary>
///  The return type of the target.
/// </summary>
public ReturnType ReturnType
{
    get
    {
        return _returnType;
    }
}

/// <summary>
/// The aspect type.
/// </summary>
public string Type
{
    get
    {
        return _type;
    }
}

/// <summary>
/// The aspect name.
/// </summary>
public string Name
{
```

```
            get
            {
                return _name;
            }
        }

        /// <summary>
        /// The aspect method.
        /// </summary>
        public Method Method
        {
            get
            {
                return _method;
            }
        }

        /// <summary>
        /// The target type.
        /// </summary>
        public string TargetType
        {
            get
            {
                return _targetType;
            }
        }

        /// <summary>
        /// A textual representation of the insert pointcut.
        /// </summary>
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append("insert ");
            sb.Append(_insertType.ToString());
            sb.Append(" ");
            sb.Append(_access.ToString());
            sb.Append(" ");
            sb.Append(_invocationKind.ToString());
            sb.Append(" ");
            sb.Append(_returnType.ToString());
            sb.Append(" ");
            sb.Append(_type);
            if(_insertType == InsertTypeEnum.DELEGATE || _insertType ==
                InsertTypeEnum.METHOD)
                sb.Append(":" + _method.ToString());
            else if(_insertType == InsertTypeEnum.FIELD || _insertType ==
                InsertTypeEnum.PROPERTY)
                sb.Append(":" + _name);
            sb.Append(" into ");
            sb.Append(_targetType);
            return sb.ToString();
        }
    }

    /// <summary>
    /// Stores information for a modify pointcut.
    /// </summary>
    public struct Modify
    {
        private string _targetType;
```

```
private ModifyTypeEnum _modifyType;
private string _inheritType;

/// <summary>
/// Creates an instance of the Modify struct.
/// </summary>
public Modify(string targetType, string inheritType)
{
    _targetType = targetType;
    _modifyType = default(ModifyTypeEnum);
    _inheritType = inheritType;
}

/// <summary>
/// Creates an instance of the Modify struct.
/// </summary>
public Modify(string targetType, ModifyTypeEnum modifyType, string
    inheritType)
{
    _targetType = targetType;
    _modifyType = modifyType;
    _inheritType = inheritType;
}

/// <summary>
/// The target type.
/// </summary>
public string TargetType
{
    get
    {
        return _targetType;
    }
}

/// <summary>
/// The modify type.
/// </summary>
public ModifyTypeEnum ModifyType
{
    get
    {
        return _modifyType;
    }
}

/// <summary>
/// The aspect type.
/// </summary>
public string InheritType
{
    get
    {
        return _inheritType;
    }
}

/// <summary>
/// A textual representation of the modify pointcut.
/// </summary>
public override string ToString()
{
```

```
            return "modify " + _targetType + " inherit " + _inheritType + ";";
        }
    }

    /// <summary>
    /// Stores information related to the returntype of a method, delegate, field
        or property.
    /// </summary>
    public struct ReturnType
    {
        private ReturnTypeEnum _type;
        private string _specificType;

        /// <summary>
        /// Creates an instance of the ReturnType struct.
        /// </summary>
        public ReturnType(ReturnTypeEnum type, string specificType)
        {
            _type = type;
            _specificType = specificType;
        }

        /// <summary>
        /// A description of the return type.
        /// </summary>
        public ReturnTypeEnum Type
        {
            get
            {
                return _type;
            }
        }

        /// <summary>
        /// The return type (for non-void and non-any return type only).
        /// </summary>
        public string SpecificType
        {
            get
            {
                return _specificType;
            }
        }

        /// <summary>
        /// A textual representation of the return type.
        /// </summary>
        public override string ToString()
        {
            if (_type == ReturnTypeEnum.ANY)
                return "*";
            else if (_type == ReturnTypeEnum.VOID)
                return "void";
            else
                return _specificType;

        }
    }

    /// <summary>
    /// Stores information for a list of arguments for a method or delegate.
    /// </summary>
```

```
public struct ArgumentList
{
    private ArgTypeEnum _arg_type;
    private ICollection<string> _arguments;

    /// <summary>
    /// Creates an instance of the ArgumentList struct.
    /// </summary>
    public ArgumentList(ArgTypeEnum arg_type, ICollection<string> arguments)
    {
        _arg_type = arg_type;
        _arguments = arguments;
    }

    /// <summary>
    ///  The type of argument.
    /// </summary>
    public ArgTypeEnum ArgumentType
    {
        get
        {
            return _arg_type;
        }
    }

    /// <summary>
    /// The collection of arguments.
    /// </summary>
    public ICollection<string> Arguments
    {
        get
        {
            return _arguments;
        }
    }
}

/// <summary>
/// Stores information for a method.
/// </summary>
public struct Method
{
    private string _name;
    private ArgumentList _argumentList;

    /// <summary>
    /// Creates an instance of the Method struct.
    /// </summary>
    public Method(string name, ArgumentList argumentList)
    {
        _name = name;
        _argumentList = argumentList;
    }

    /// <summary>
    /// The name of the method.
    /// </summary>
    public string Name
    {
        get
        {
            return _name;
```

```
            }
        }

        /// <summary>
        /// The list of arguments.
        /// </summary>
        public ArgumentList ArgumentList
        {
            get
            {
                return _argumentList;
            }
        }

        /// <summary>
        /// A textual representation of the method.
        /// </summary>
        public override string ToString()
        {
            if (_argumentList.ArgumentType == ArgTypeEnum.ANY)
                return _name + "(*)";
            else if (_argumentList.ArgumentType == ArgTypeEnum.SPECIFIC)
            {
                string output = "";
                bool first = true;
                foreach (string argument in _argumentList.Arguments)
                {
                    if (!first)
                        output += ",";

                    first = false;
                    output += argument;
                }
                return _name + "(" + output + ")";
            }
            else
                return _name + "()";
        }
    }

    /// <summary>
    /// Stores information for an inheritance specification.
    /// </summary>
    public struct Inherit
    {
        private InheritTypeEnum _inheritType;
        private ModifyTypeEnum _modifyType;
        private string _specificType;

        /// <summary>
        /// Creates an instance of the Inherit struct.
        /// </summary>
        public Inherit(InheritTypeEnum inheritType, string specificType)
        {
            _inheritType = inheritType;
            _modifyType = default(ModifyTypeEnum);
            _specificType = specificType;
        }

        /// <summary>
        /// Creates an instance of the Inherit struct.
        /// </summary>
```

```
        public Inherit(InheritTypeEnum inheritType, ModifyTypeEnum modifyType,
            string specificType)
        {
            _inheritType = inheritType;
            _modifyType = modifyType;
            _specificType = specificType;
        }

        /// <summary>
        /// The type of inheritance specification.
        /// </summary>
        public InheritTypeEnum InheritType
        {
            get
            {
                return _inheritType;
            }
        }

        public ModifyTypeEnum ModifyType
        {
            get
            {
                return _modifyType;
            }
        }

        /// <summary>
        /// The specific type (if any).
        /// </summary>
        public string SpecificType
        {
            get
            {
                return _specificType;
            }
        }

        /// <summary>
        /// A textual representation of the inheritance specification.
        /// </summary>
        public override string ToString()
        {
            if (_inheritType == InheritTypeEnum.ANY)
                return "inherits *";
            else
                return "inherits " + _specificType;
        }
    }

    /// <summary>
    /// Interface for parsing an input file.
    /// </summary>
    public interface IParser
    {
        void Parse();
        ICollection<Around> AroundStatements { get; }
        ICollection<Insert> InsertStatements { get; }
        ICollection<Modify> ModifyStatements { get; }
    }

    /// <summary>
```

```
/// Parses an input file and creates Around, Insert and Modify structs for
    describing the pointcuts found.
/// </summary>
public class Parser : IParser
{
    private LexicalAnalysis.Scanner _scanner;
    private ICollection<Around> _aroundStatements;
    private ICollection<Insert> _insertStatements;
    private ICollection<Modify> _modifyStatements;
    private int _statementNumber = 0;

    /// <summary>
    /// Creates an instance of the parser.
    /// </summary>
    /// <param name="scanner">A scanner instance that can be used for
        retrieving tokens from the input file.</param>
    public Parser(LexicalAnalysis.Scanner scanner)
    {
        _scanner = scanner;
        _aroundStatements = new LinkedList<Around>();
        _insertStatements = new LinkedList<Insert>();
        _modifyStatements = new LinkedList<Modify>();
    }

    /// <summary>
    /// A grammar method. Parses the first token found.
    /// </summary>
    protected void Start()
    {
        _statementNumber++;

        switch (_scanner.Token)
        {
            case LexicalAnalysis.Scanner.TokenType.AROUND:
                _scanner.Next();
                Around();
                break;
            case LexicalAnalysis.Scanner.TokenType.INSERT:
                _scanner.Next();
                Insert();
                break;
            case LexicalAnalysis.Scanner.TokenType.MODIFY:
                _scanner.Next();
                Modify();
                break;
            default:
                throw new Exceptions.ParseError("Expected 'around', 'insert'
                    or 'modify', but found " + TokenAsString());
        }

        if (_scanner.Token != LexicalAnalysis.Scanner.TokenType.EOF) // more
            tokens are available - continue
            Start(); // parse the next statement
    }

    /// <summary>
    /// A grammer method. Parses an around statement.
    /// </summary>
    protected void Around()
    {
        AccessEnum access = Access(); // parse the access specified
        InvocationKindEnum invocationKind = InvocationKind(); // parse the
```

```
            invocation kind (instance, static, *)
        ReturnType returnType = ReturnType(); // parse the return type
        string type = Name(); // parse the target type (namespace + class name
            )
        CheckColon();
        Method method = Method(); // parse the method (including any specified
             arguments)
        Inherit inherit = Inherit(); // parse the inherits specification (if
            any)
        CheckDo();
        string adviceType = ForcedName(); // parse the advice type (namespace
            + class name)
        CheckColon();
        string adviceMethod = ForcedName(); // parse the advice method
        CheckEOS();

        // store this around statement
        _aroundStatements.Add(new Around(access, invocationKind, returnType,
            type, method, inherit, adviceType, adviceMethod));
    }

    /// <summary>
    /// A grammar method. Parses an insert statement.
    /// </summary>
    protected void Insert()
    {
        Insert insert = Introduction();
        CheckEOS();
        _insertStatements.Add(insert);
    }

    /// <summary>
    /// A grammar method. Parses a modify statement.
    /// </summary>
    protected void Modify()
    {
        string targetType = ForcedName();
        Inherit inherit = ForcedInherit();
        CheckEOS();

        _modifyStatements.Add(new Modify(targetType, inherit.ModifyType,
            inherit.SpecificType));
    }

    /// <summary>
    /// A grammar method. Parses an access specifier (which is optional).
    /// </summary>
    /// <returns>An AccessEnum that describes the access specifier found.</
        returns>
    protected AccessEnum Access()
    {
        switch (_scanner.Token)
        {
            case LexicalAnalysis.Scanner.TokenType.PUBLIC:
                _scanner.Next(); // fetch next token
                return AccessEnum.PUBLIC;
            case LexicalAnalysis.Scanner.TokenType.PRIVATE:
                _scanner.Next(); // fetch next token
                return AccessEnum.PRIVATE;
            case LexicalAnalysis.Scanner.TokenType.PROTECTED:
                _scanner.Next(); // fetch next token
                return AccessEnum.PROTECTED;
```

```
            case LexicalAnalysis.Scanner.TokenType.INTERNAL:
                _scanner.Next(); // fetch next token
                return AccessEnum.INTERNAL;
            case LexicalAnalysis.Scanner.TokenType.ANY:
                _scanner.Next(); // fetch next token
                return AccessEnum.ANY;
            default:
                throw new Exceptions.ParseError("Expected 'public', 'private',
                    'protected', 'internal' or '*', but found " +
                    TokenAsString());
        }
    }

    /// <summary>
    /// A grammar method. Parses an introduction statement.
    /// </summary>
    /// <returns>An Insert struct describing the insert statement.</returns>
    protected Insert Introduction()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.CLASS ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.INTERFACE ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.ENUM ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.ATTRIBUTE)
            return TypeIntro();
        else if(_scanner.Token == LexicalAnalysis.Scanner.TokenType.PROPERTY
            || _scanner.Token == LexicalAnalysis.Scanner.TokenType.FIELD ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.EVENT ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.METHOD ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.DELEGATE)
            return MemberIntro();
        else
            throw new Exceptions.ParseError("Expected 'class', 'interface', '
                property', 'field', 'event', 'method' or 'delegate', but found
                 " + TokenAsString());
    }

    /// <summary>
    /// A grammar method. Parses a typeintro (class or interface).
    /// </summary>
    /// <returns>An insert struct describing the insert statement.</returns>
    protected Insert TypeIntro()
    {
        InsertTypeEnum insertType;
        insertType = TypeConstruct();

        string type = ForcedName(); // parse the type
        CheckInto();
        string targetType = Name(); // parse the target type
        return new Insert(insertType, type, targetType);
    }

    /// <summary>
    /// A grammar method. Check if the current token is a valid type (class,
        enum or attribute).
    /// </summary>
    /// <returns>An InsertTypeEnum describing the kind of type specified.</
        returns>
    protected InsertTypeEnum TypeConstruct()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.CLASS ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.INTERFACE)
        {
```

```
            _scanner.Next();
            return InsertTypeEnum.CLASS;
        }
        else if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.ATTRIBUTE
            )
        {
            _scanner.Next();
            return InsertTypeEnum.ATTRIBUTE;
        }
        else if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.ENUM)
        {
            _scanner.Next();
            return InsertTypeEnum.ENUM;
        }
        else // invalid type of construct specified by the user
            throw new Exceptions.ParseError("Expected 'class', 'interface', '
                enum' or 'attribute', but found " + TokenAsString());
    }

    /// <summary>
    /// A grammar method. Parses a memberintro (method, delegate, field of
        property).
    /// </summary>
    /// <returns>An insert struct describing the insert statement.</returns>
    protected Insert MemberIntro()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.PROPERTY ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.FIELD ||
            _scanner.Token == LexicalAnalysis.Scanner.TokenType.EVENT) // this
             is a property, field or event insert statement
        {
            InsertTypeEnum insertType = MemberConstruct();
            AccessEnum access = Access();
            InvocationKindEnum invocationKind = InvocationKind();
            ReturnType returnType = ReturnType();
            string type = ForcedName();
            CheckColon();
            string name = ForcedName();
            CheckInto();
            string targetType = Name();

            return new Insert(insertType, access, invocationKind, returnType,
                type, name, targetType);
        }
        else
        {
            InsertTypeEnum insertType = MemberConstructWithArg();
            AccessEnum access = Access();
            InvocationKindEnum invocationKind = InvocationKind();
            ReturnType returnType = ReturnType();
            string type = ForcedName();
            CheckColon();
            Method method = ForcedMethod();
            CheckInto();
            string targetType = Name();

            return new Insert(insertType, access, invocationKind, returnType,
                type, method, targetType);
        }
    }

    /// <summary>
```

```
/// A grammar method. Parses a memberconstruct (either "field", "property
    " or "event").
/// </summary>
/// <returns>An InsertTypeEnum describing the kind of construct found.</
    returns>
protected InsertTypeEnum MemberConstruct()
{
    if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.PROPERTY)
    {
        _scanner.Next();
        return InsertTypeEnum.PROPERTY;
    }
    else if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.FIELD)
    {
        _scanner.Next();
        return InsertTypeEnum.FIELD;
    }
    else if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.EVENT)
    {
        _scanner.Next();
        return InsertTypeEnum.EVENT;
    }
    else
        throw new Exceptions.ParseError("Expected 'property', 'event' or '
            field', but found " + TokenAsString());
}

/// <summary>
/// A grammar method. Parses a memberconstructwitharg (either "method" or
    "delegate").
/// </summary>
/// <returns>An InsertTypeEnum describing the kind of construct found.</
    returns>
protected InsertTypeEnum MemberConstructWithArg()
{
    if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.METHOD)
    {
        _scanner.Next();
        return InsertTypeEnum.METHOD;
    }
    else if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.DELEGATE)
    {
        _scanner.Next();
        return InsertTypeEnum.DELEGATE;
    }
    else
        throw new Exceptions.ParseError("Expected 'method' or 'delegate',
            but found " + TokenAsString());
}

/// <summary>
/// A grammar method. Parses an invocation kind (static, instance or *).
    The invocation kind is optional.
/// </summary>
/// <returns>An InvocationKindEnum describing the invocation kind.</
    returns>
protected InvocationKindEnum InvocationKind()
{
    switch (_scanner.Token)
    {
        case LexicalAnalysis.Scanner.TokenType.STATIC:
            _scanner.Next(); // fetch next token
```

```
                return InvocationKindEnum.STATIC;
            case LexicalAnalysis.Scanner.TokenType.INSTANCE:
                _scanner.Next(); // fetch next token
                return InvocationKindEnum.INSTANCE;
            case LexicalAnalysis.Scanner.TokenType.ANY:
                _scanner.Next(); // fetch next token
                return InvocationKindEnum.ANY;
            default:
                throw new Exceptions.ParseError("Expected 'static', 'instance'
                    or '*', but found " + TokenAsString());
        }
    }

    /// <summary>
    /// A grammar method. Parses the return type.
    /// </summary>
    /// <returns>A ReturnType struct describing the return type specified by
        the user. There are 3 kinds of return types: void, any (*) or specific
         (any type).</returns>
    protected ReturnType ReturnType()
    {
        switch (_scanner.Token)
        {
            case LexicalAnalysis.Scanner.TokenType.VOID:
                _scanner.Next(); // fetch next token
                return new ReturnType(ReturnTypeEnum.VOID, "");
            case LexicalAnalysis.Scanner.TokenType.ANY:
                _scanner.Next(); // fetch next token
                return new ReturnType(ReturnTypeEnum.ANY, "");
            case LexicalAnalysis.Scanner.TokenType.NAME:
                return new ReturnType(ReturnTypeEnum.SPECIFIC, Name()); //
                    parse the return type
            default:
                throw new Exceptions.ParseError("Expected return type, but
                    found " + TokenAsString());
        }
    }

    /// <summary>
    /// A grammar method. Parses a name. '*' is allowed.
    /// </summary>
    /// <returns>A string containing the name.</returns>
    protected string Name()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.NAME)
        {
            string sval = _scanner.Sval;
            int indexAny = sval.IndexOf("*");
            if (indexAny != -1 && !(indexAny == 0 || indexAny == sval.Length -
                1)) // the * is located is not located at either end of the
                string - this is not allowed
                throw new Exceptions.ParseError("The wildcard character (*) is
                    only allowed in the beginning or at the end of a name.");

            if (sval.IndexOf("*", indexAny + 1) != -1) // more than one * is
                specified - this is not allowed
                throw new Exceptions.ParseError("Only one wildcard character
                    (*) is allowed within a name.");

            _scanner.Next();
            return sval;
        }
```

```
        else if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.ANY)
        {
            _scanner.Next();
            return "*";
        }
        else
            throw new Exceptions.ParseError("Expected name, but found " +
                TokenAsString());
    }

    /// <summary>
    /// A grammar method. Parses a name. '*' is NOT allowed.
    /// </summary>
    /// <returns>A string containing the name.</returns>
    protected string ForcedName()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.NAME) // check
             that the "." is followed by a name
            if (!(_scanner.Sval.Equals("*")))
            {
                string sval = _scanner.Sval;
                _scanner.Next();
                return sval;
            }
            else
                throw new Exceptions.ParseError("Expected name, but found '*'"
                    );
        else
            throw new Exceptions.ParseError("Expected name, but found " +
                TokenAsString());
    }

    /// <summary>
    /// A grammar method. Parses a method. '*' is allowed for both method name
         and arguments.
    /// </summary>
    /// <returns>A Method struct describing the method.</returns>
    protected Method Method()
    {
        string method_name = Name(); // parse the name of the method
        CheckParOpen();
        ArgumentList argumentlist = Arglist(); // parse the list of method
            arguments
        CheckParClose();
        return new Method(method_name, argumentlist);
    }

    /// <summary>
    /// A grammar method. Parses a method. '*' is NOT allowed.
    /// </summary>
    /// <returns>A Method struct describing the method.</returns>
    protected Method ForcedMethod()
    {
        string methodName = ForcedName();
        CheckParOpen();
        ArgumentList argumentlist = ForcedArgList();
        CheckParClose();
        return new Method(methodName, argumentlist);
    }

    /// <summary>
    /// A grammar method. Parses a list of arguments for a method. '*' is
```

```
    allowed.
/// </summary>
/// <returns>An ArgumentList struct describing the arguments.</returns>
protected ArgumentList Arglist()
{
    if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.ANY)
    {
        _scanner.Next();
        return new ArgumentList(ArgTypeEnum.ANY, null);
    }
    else
        return ForcedArgList();
}

/// <summary>
/// A grammar method. Parses a list of arguments for a method. '*' is NOT
    allowed.
/// </summary>
/// <returns>An ArgumentList struct describing the arguments.</returns>
protected ArgumentList ForcedArgList()
{
    if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.PARCLOSE) //
        no arguments specified
        return new ArgumentList(ArgTypeEnum.NONE, null);
    else
    {
        string name = ForcedName();
        LinkedList<string> arguments = ArgListOpt();
        arguments.AddFirst(name);
        return new ArgumentList(ArgTypeEnum.SPECIFIC, arguments);
    }
}

/// <summary>
/// A grammar method. Parses the optional part of the list of arguments (
    the arguments following the very first argument).
/// </summary>
/// <returns>A LinkedList instance containing all arguments found.</
    returns>
protected LinkedList<string> ArgListOpt()
{
    if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.COMMA)
    {
        _scanner.Next();
        string name = Name();
        LinkedList<string> arguments = ArgListOpt();
        arguments.AddFirst(name);
        return arguments;
    }
    else // final argument has been reached
        return new LinkedList<string>();
}

/// <summary>
/// A grammar method. Parses the inherits statement (which is optional).
/// </summary>
/// <returns>An Inherit struct describing the inheritance requirement
    specified. If no inherits statement is specified, ANY (*) is returned
    .</returns>
protected Inherit Inherit()
{
    if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.INHERITS)
```

```
        {
            _scanner.Next();
            return new Inherit(InheritTypeEnum.SPECIFIC, Name());
        }
        else
            return new Inherit(InheritTypeEnum.ANY, "");
    }

    /// <summary>
    /// A grammar method. Parses the inherits statement (NOT optional).
    /// </summary>
    /// <returns>An Inherit struct describing the inheritance requirement
        specified.</returns>
    protected Inherit ForcedInherit()
    {
        ModifyTypeEnum modifyType;
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.INHERITS)
            modifyType = ModifyTypeEnum.INHERIT;
        else if(_scanner.Token == LexicalAnalysis.Scanner.TokenType.IMPLEMENT)
            modifyType = ModifyTypeEnum.IMPLEMENT;
        else
            throw new Exceptions.ParseError("Expected 'inherit' or 'implement
                ', but found " + TokenAsString());

        _scanner.Next();
        return new Inherit(InheritTypeEnum.SPECIFIC, modifyType, ForcedName())
            ;
    }

    /// <summary>
    /// A helper method. Checks that the next token is a DO.
    /// </summary>
    protected void CheckDo()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.DO)
            _scanner.Next();
        else
            throw new Exceptions.ParseError("Expected 'do', but found " +
                TokenAsString());
    }

    /// <summary>
    /// A helper method. Checks that the next token is COLON.
    /// </summary>
    protected void CheckColon()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.COLON)
            _scanner.Next();
        else
            throw new Exceptions.ParseError("Expected ':', but found " +
                TokenAsString());
    }

    /// <summary>
    /// A helper method. Checks that the next token is a PAROPEN.
    /// </summary>
    protected void CheckParOpen()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.PAROPEN)
            _scanner.Next();
        else
            throw new Exceptions.ParseError("Expected '(', but found " +
```

```
            TokenAsString());
    }

    /// <summary>
    /// A helper method. Checks that the next token is a PARCLOSE.
    /// </summary>
    protected void CheckParClose()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.PARCLOSE)
            _scanner.Next();
        else
            throw new Exceptions.ParseError("Expected ')', but found " +
                TokenAsString());
    }

    /// <summary>
    /// A helper method. Checks that the next token is a INTO.
    /// </summary>
    protected void CheckInto()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.INTO)
            _scanner.Next();
        else
            throw new Exceptions.ParseError("Expected 'into', but found " +
                TokenAsString());
    }

    /// <summary>
    /// A helper method. Checks that the next token is a EOS (End-Of-Statement
        ).
    /// </summary>
    protected void CheckEOS()
    {
        if (_scanner.Token == LexicalAnalysis.Scanner.TokenType.EOS)
            _scanner.Next();
        else
            throw new Exceptions.ParseError("Expected ';', but found " +
                TokenAsString());
    }

    /// <summary>
    /// Parses the input file specifed. After invoking this method, the output
         is available via the properties AroundStatement, InsertStatement and
        ModifyStatements.
    /// </summary>
    public void Parse()
    {
        Start();

        if (_scanner.Token != LexicalAnalysis.Scanner.TokenType.EOF)
            throw new Exceptions.ParseError("Expected end of file");
    }

    /// <summary>
    /// Returns the list of around statements found. It makes no sense to get
        this list before invoking Parse().
    /// </summary>
    public ICollection<Around> AroundStatements
    {
        get
        {
            return _aroundStatements;
```

```
        }
    }

    /// <summary>
    /// Returns the list of insert statements found. It makes no sense to get
        this list before invoking Parse().
    /// </summary>
    public ICollection<Insert> InsertStatements
    {
        get
        {
            return _insertStatements;
        }
    }

    /// <summary>
    /// Returns the list of modify statements found. It makes no sense to get
        this list before invoking Parse().
    /// </summary>
    public ICollection<Modify> ModifyStatements
    {
        get
        {
            return _modifyStatements;
        }
    }

    /// <summary>
    /// Returns the current token in a human-readable form.
    /// </summary>
    /// <returns>A string representation of the current token.</returns>
    private string TokenAsString()
    {
        string output;
        switch (_scanner.Token)
        {
            case LexicalAnalysis.Scanner.TokenType.ANY:
                output = "'*'";
                break;
            case LexicalAnalysis.Scanner.TokenType.COLON:
                output = "':'";
                break;
            case LexicalAnalysis.Scanner.TokenType.COMMA:
                output = "','";
                break;
            case LexicalAnalysis.Scanner.TokenType.EOF:
                output = "end-of-file";
                break;
            case LexicalAnalysis.Scanner.TokenType.EOS:
                output = "';'";
                break;
            case LexicalAnalysis.Scanner.TokenType.NAME:
                output = "'" + _scanner.Sval + "'";
                break;
            case LexicalAnalysis.Scanner.TokenType.PARCLOSE:
                output = "')'";
                break;
            case LexicalAnalysis.Scanner.TokenType.PAROPEN:
                output = "'('";
                break;
            default:
                output = "'" + _scanner.Token.ToString().ToLower() + "'";
```

```
                    break;
                }

                return output + " (statement " + _statementNumber + ")";
            }

        }
}
```

Exceptions.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.Pointcut.Exceptions
{
    /// <summary>
    /// Exception thrown when the pointcut file could not be found.
    /// </summary>
    public class InputFileNotFoundException : Exception
    {
        public InputFileNotFoundException(string message)
            : base(message)
        {
        }
    }

    /// <summary>
    /// Exception thrown when the parsed experienced an error.
    /// </summary>
    public class ParseError : Exception
    {
        public ParseError(string message)
            : base(message)
        {
        }
    }
}
```

Appendix W

Source code for YIIHAW - Controller

InsertHandler.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAW.Pointcut;
using Mono.Cecil;
using YIIHAW.Weaver;

namespace YIIHAW.Controller
{
    /// <summary>
    /// Handles the insertion of contructs from an aspect assembly to an target
        assembly.
    /// </summary>
    public class InsertHandler
    {
        private AssemblyDefinition _aspectAssembly;
        private AssemblyDefinition _targetAssembly;
        private LocalMapperCollection _localMaps;
        private GlobalMapperCollection _globalMaps;

        private Introduction _weaver;

        /// <summary>
        /// Creates a new InsertHandler.
        /// </summary>
        /// <param name="aspectAssembly">The aspect assembly where the constructs
            that should be inserted are located.</param>
        /// <param name="targetAssembly">The target assembly where the constructs
            will get inserted.</param>
        /// <param name="localMaps">A LocalMapperCollection which will be used
            throughout the insertions.</param>
        /// <param name="globalMaps">A GlobalMapperCollection which will be used
            throughout the insertions.</param>
        public InsertHandler(AssemblyDefinition aspectAssembly, AssemblyDefinition
             targetAssembly, LocalMapperCollection localMaps,
            GlobalMapperCollection globalMaps)
        {
            _localMaps = localMaps;
            _globalMaps = globalMaps;
            _aspectAssembly = aspectAssembly;
```

```
        _targetAssembly = targetAssembly;
        _weaver = new Introduction();
    }

    /// <summary>
    /// Processes an insert statement by checking the type of contruct to
        insert,
    /// and taking the needed actions to insert the construct.
    /// </summary>
    /// <param name="insert">The pointcut represented as an insert statement
        .</param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
         of the insertion.</param>
    public void ProcessStatement(Insert insert, bool firstPass)
    {
        switch (insert.InsertType)
        {
            case InsertTypeEnum.METHOD:
                MethodDefinition aspect = FindMethod(insert);
                InsertAspectMethod(aspect, insert, firstPass, true);
                break;
            case InsertTypeEnum.PROPERTY:
                PropertyDefinition property = FindProperty(insert);
                InsertAspectProperty(property, insert, firstPass);
                break;
            case InsertTypeEnum.FIELD:
                FieldDefinition field = FindField(insert);
                InsertAspectField(field, insert, firstPass);
                break;
            case InsertTypeEnum.CLASS:
                TypeDefinition type = FindType(insert);
                InsertAspectType(type, insert, firstPass);
                break;
            case InsertTypeEnum.EVENT:
                EventDefinition eventDef = FindEvent(insert);
                InsertAspectEvent(eventDef, insert, firstPass);
                break;
        }
    }

    /// <summary>
    /// Inserts an event from the aspect assembly and into the target assembly
        .
    /// This method also inserts the field and methods beloging to the event.
    /// </summary>
    /// <param name="aspectEvent">The event from the aspect assembly to insert
        .</param>
    /// <param name="insert">The insert statement indicating where the event
        should be inserted.</param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
         of the insertion.</param>
    private void InsertAspectEvent(EventDefinition aspectEvent, Insert insert,
         bool firstPass)
    {
        foreach (TypeDefinition typeDef in _targetAssembly.MainModule.Types)
        {
            // ignore all interfaces and modules
            if (typeDef.FullName.Equals("<Module>") || typeDef.IsInterface)
                continue;

            // ignore newly inserted types
            if (_globalMaps.Types.ContainsValue(typeDef))
```

```
                continue;

            // check that this target matches the type defined in the pointcut
                specification
            if (!Helper.CheckTarget(insert.TargetType, typeDef.FullName))
                continue;

            // check if an event with the same name is already in the class.
            foreach (EventDefinition targetEventDef in typeDef.Events)
                if (targetEventDef.Name.Equals(aspectEvent.Name))
                    throw new Exceptions.IllegalOperationException("Event '" +
                        aspectEvent.Name + "' can not be inserted into class
                        '" + typeDef.FullName + "', as an event with the same
                        name already exist in this class.");

            if (firstPass)
                YIIHAW.Output.OutputFormatter.AddEvent(aspectEvent.
                    DeclaringType.FullName + "." + aspectEvent.Name, typeDef.
                    FullName);

            // find and insert the field belonging to the event.
            FieldDefinition eventField = FindField(insert);
            InsertField(eventField, typeDef, firstPass);

            // insert any methods belonging to the event.
            if (aspectEvent.AddMethod != null)
                InsertMethod(aspectEvent.AddMethod, typeDef, firstPass, false)
                    ;

            if (aspectEvent.InvokeMethod != null)
                InsertMethod(aspectEvent.InvokeMethod, typeDef, firstPass,
                    false);

            if (aspectEvent.RemoveMethod != null)
                InsertMethod(aspectEvent.RemoveMethod, typeDef, firstPass,
                    false);

            // insert the actual event.
            InsertEvent(aspectEvent, typeDef, firstPass);
        }
    }

    /// <summary>
    /// Inserts an event into a given target.
    /// </summary>
    /// <param name="aspectEvent">The event from the aspect assembly to insert
        .</param>
    /// <param name="typeDef">The type where the event should be inserted.</
        param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
         of the insertion.</param>
    private void InsertEvent(EventDefinition aspectEvent, TypeDefinition
        typeDef, bool firstPass)
    {

        if (!firstPass)
        {
            EventDefinition newEvent = aspectEvent.Clone();

            newEvent.CustomAttributes.Clear(); // The attributes set by Clone
                () are not correct.
```

```
// update method references to the local methods in the target
    assembly.
if (aspectEvent.AddMethod != null)
    newEvent.AddMethod = _localMaps.Methods.Lookup(typeDef,
        aspectEvent.AddMethod);

if (aspectEvent.InvokeMethod != null)
    newEvent.InvokeMethod = _localMaps.Methods.Lookup(typeDef,
        aspectEvent.InvokeMethod);

if (aspectEvent.RemoveMethod != null)
    newEvent.RemoveMethod = _localMaps.Methods.Lookup(typeDef,
        aspectEvent.RemoveMethod);

// update the reference to the events type.
if (aspectEvent.EventType.Scope == aspectEvent.DeclaringType.Scope
    )
{
    GlobalMapperEntry<TypeReference> typeEntry = _globalMaps.
        TypeReferences.Lookup(aspectEvent.EventType);
    if (typeEntry == null)
        throw new Exceptions.IllegalOperationException("Unable to
            locate the class '" + aspectEvent.EventType.FullName +
            "' in the target assembly. This class is used by the
            event '" + aspectEvent.Name + "', which you have
            specified should be inserted into the target assembly.
             Please specify that the class '" + aspectEvent.
            EventType.FullName + "' should be introduced as well
            using the pointcut file.");
    else if (typeEntry.IsAmbiguousReference)
        throw new Exceptions.IllegalOperationException("Unable to
            locate the class '" + aspectEvent.EventType.FullName +
            "', which is used by the event '" + aspectEvent.Name
            + "'. The class '" + aspectEvent.EventType.FullName +
            "' is inserted at multiple locations in the target
            assembly (thus causing the reference to this class to
            be ambiguous).");
    else
        newEvent.EventType = typeEntry.Reference;
}
else
{
    if (!(YIIHAW.Weaver.Helper.IsAssemblyInRefs(typeDef.Module.
        AssemblyReferences, aspectEvent.EventType) || YIIHAW.
        Weaver.Helper.IsAssemblyTarget(aspectEvent.EventType,
        typeDef)))
        YIIHAW.Output.OutputFormatter.AddWarning("It is not
            possible to type check the usage of '" + aspectEvent.
            EventType.FullName + "'. Please make sure that this
            class is available from the target assembly.");

    if (YIIHAW.Weaver.Helper.IsAssemblyTarget(aspectEvent.
        EventType, typeDef))
        newEvent.EventType = YIIHAW.Weaver.Helper.FindLocalType(
            typeDef, aspectEvent.EventType);
    else
        newEvent.EventType = typeDef.Module.Import(aspectEvent.
            EventType);
}

// insert attributes
foreach (CustomAttribute attribute in aspectEvent.CustomAttributes
```

```
                )
                newEvent.CustomAttributes.Add(_weaver.CopyAndUpdateAttribute(
                    aspectEvent.DeclaringType, typeDef, _globalMaps, attribute
                    ));

            // add the event to the target, and update the target assembly.
            typeDef.Events.Add(newEvent);
            typeDef.Module.Import(typeDef);

        }
    }

    /// <summary>
    /// Finds an event that should be inserted from the aspect assembly.
    /// </summary>
    /// <param name="insert">The insert statement, which indicates what event
        to insert.</param>
    /// <returns>An EventDefinition representing the event in aspect assembly
        that should be inserted.</returns>
    private EventDefinition FindEvent(Insert insert)
    {
        foreach (TypeDefinition typeDef in _aspectAssembly.MainModule.Types)
        {
            if (typeDef.FullName.Equals(insert.Type)) // we have found the
                correct class - run through all events in the class
            {
                foreach (EventDefinition eventDef in typeDef.Events)
                {
                    // check the event name
                    if (!eventDef.Name.Equals(insert.Name))
                        continue;

                    // check the event type
                    if (!(insert.ReturnType.Type == YIIHAW.Pointcut.
                        ReturnTypeEnum.ANY) && (!Helper.IsTypeEqual(insert.
                        ReturnType.SpecificType, eventDef.EventType)))
                        continue;

                    //check the access specifier - this can only be done on
                        the events methods.
                    if (eventDef.AddMethod != null && !Helper.
                        CheckAccessSpecifier(eventDef.AddMethod.Attributes,
                        insert.Access))
                        continue;

                    return eventDef;
                }
            }
        }
        throw new Exceptions.ConstructNotFoundException("The event '" + insert
            .Name + "' could not be found in class " + insert.Type + ".");
    }


    /// <summary>
    /// Inserts an type from the aspect assembly and into the target assembly.
    /// </summary>
    /// <param name="type">The type from the aspect assembly to insert.</param
        >
    /// <param name="insert">The insert statement indicating where the type
        should be inserted.</param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
```

```
        of the insertion.</param>
    private void InsertAspectType(TypeDefinition type, Insert insert, bool
        firstPass)
    {
        List<string> namespacesInsertedInto = new List<string>(); // keeps
            control of the namespaces that the type has been inserted into.
        bool found = false;

        foreach (TypeDefinition typeDef in _targetAssembly.MainModule.Types)
        {
            // avoiding the namespace of newly inserted types.
            if (typeDef.Namespace == "")
                continue;

            // check if the insert statement matches the namespace of the
                target.
            if (Helper.CheckTarget(insert.TargetType, typeDef.Namespace))
            {
                if (firstPass)
                {
                    if (namespacesInsertedInto.Contains(typeDef.Namespace)) //
                        type has already been inserted into the namespace
                        continue;

                    TypeDefinition newTypeDef = new TypeDefinition(type.Name,
                        typeDef.Namespace, type.Attributes, type.BaseType);
                    newTypeDef.Attributes = MakeTypeUnNested(newTypeDef.
                        Attributes); // if the inserted type was nested in
                        aspect in should be unnested now.
                    InsertType(type, firstPass, newTypeDef); // do the actual
                        insertion of the type.
                    namespacesInsertedInto.Add(typeDef.Namespace);
                    Output.OutputFormatter.AddType(type.Name, typeDef.
                        Namespace);
                    found = true;
                }
                else if (type.Name.Equals(typeDef.Name)) // second pass
                {
                    InsertType(type, firstPass, typeDef);
                    found = true;
                }

            }
            // If the insert did not match the namespace of the target type,
                it might match the actually type.
            // in which case is should be inserted as a nested type. This is
                only possible if the insert statemet
            // contains a fully qualified name of the target.
            // The change of '/' to '.' is done, as nested types are named
                typename/nestedtypename in Cecil.
            else if (!insert.TargetType.Contains("*") && Helper.CheckTarget(
                insert.TargetType, typeDef.FullName.Replace('/', '.')) && !
                typeDef.IsInterface) //There was not a namespace match but a
                precise typename instead, the type will be inserted as a
                nestedtype.
            {
                if (firstPass)
                {
                    TypeDefinition newTypeDef = new TypeDefinition(type.Name,
                        "", type.Attributes, type.BaseType);
                    newTypeDef.Attributes = MakeTypeNested(newTypeDef.
                        Attributes); // the type should be nested now.
```

```
                    typeDef.NestedTypes.Add(newTypeDef);
                    InsertType(type, firstPass, newTypeDef); // do the actual
                        insetion of the type.
                    Output.OutputFormatter.AddType(type.Name, typeDef.FullName
                        );
                }
                else // second pass
                    foreach (TypeDefinition nestedType in typeDef.NestedTypes)
                        if (nestedType.Name.Equals(type.Name))
                            InsertType(type, firstPass, nestedType);

                found = true;

            }
        }
        if (firstPass && !found)
            Output.OutputFormatter.AddWarning("The namespace or type '" +
                insert.TargetType + "' was not found in the target assembly.
                The type '" + type.Name + "' has not been inserted into the
                target assembly.");
    }

    /// <summary>
    /// Changes the attributes of a type, so that it becomes nested.
    /// </summary>
    /// <param name="typeAttributes">The type's attributes.</param>
    /// <returns>The type attributes changed so that the type is nested.</
        returns>
    private TypeAttributes MakeTypeNested(TypeAttributes typeAttributes)
    {
        if ((typeAttributes & TypeAttributes.Public) == TypeAttributes.Public)
            return typeAttributes | TypeAttributes.NestedPublic;
        else
            return typeAttributes | TypeAttributes.NestedAssembly;
    }

    /// <summary>
    /// Changes any attributes that a type may have, so that it no longer is
        nested.
    /// If the type is not nested, the attributes returned will be the same as
         the ones given as argument.
    /// </summary>
    /// <param name="typeAttributes">The type's attributes.</param>
    /// <returns>The type attributes changed so that the type is no longer
        nested.</returns>
    private TypeAttributes MakeTypeUnNested(TypeAttributes typeAttributes)
    {
        if ((typeAttributes & TypeAttributes.NestedPublic) == TypeAttributes.
            NestedPublic)
            return (typeAttributes ^ TypeAttributes.NestedPublic) |
                TypeAttributes.Public;
        else if ((typeAttributes & TypeAttributes.NestedAssembly) ==
            TypeAttributes.NestedAssembly)
            return (typeAttributes ^ TypeAttributes.NestedAssembly) |
                TypeAttributes.NotPublic;
        else if ((typeAttributes & TypeAttributes.NestedFamANDAssem) ==
            TypeAttributes.NestedFamANDAssem)
            return (typeAttributes ^ TypeAttributes.NestedFamANDAssem) |
                TypeAttributes.NotPublic;
        else if ((typeAttributes & TypeAttributes.NestedFamily) ==
            TypeAttributes.NestedFamily)
            return (typeAttributes ^ TypeAttributes.NestedFamily) |
```

```
            TypeAttributes.NotPublic;
        else if ((typeAttributes & TypeAttributes.NestedFamORAssem) ==
            TypeAttributes.NestedFamORAssem)
            return (typeAttributes ^ TypeAttributes.NestedFamORAssem) |
                TypeAttributes.NotPublic;
        else if ((typeAttributes & TypeAttributes.NestedPrivate) ==
            TypeAttributes.NestedPrivate)
            return (typeAttributes ^ TypeAttributes.NestedPrivate) |
                TypeAttributes.NotPublic;
        else return typeAttributes;
    }

    /// <summary>
    /// Inserts all the content of an aspect type into a target type.
    /// </summary>
    /// <param name="type">The aspect type to insert from.</param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
         of the insertion.</param>
    /// <param name="typeDef">The target type to insert into.</param>
    private void InsertType(TypeDefinition type, bool firstPass,
        TypeDefinition typeDef)
    {
        if (firstPass)
        {
            // recursivly insert the nested types
            foreach (TypeDefinition nestedTypeDef in type.NestedTypes)
            {
                TypeDefinition newNestedType = new TypeDefinition(
                    nestedTypeDef.Name, nestedTypeDef.Namespace, nestedTypeDef
                    .Attributes, nestedTypeDef.BaseType);
                typeDef.NestedTypes.Add(newNestedType);
                InsertType(nestedTypeDef, firstPass, newNestedType);
            }

            _targetAssembly.MainModule.Types.Add(typeDef);

            _globalMaps.Types.Add(type, typeDef);
            _globalMaps.TypeReferences.Add(type, typeDef);



        }
        else if (type.Name.Equals(typeDef.Name)) //this is the second pass and
             we have the correct type (the type inserted in the first pass).
        {
            // recursivly insert the nested types.
            foreach (TypeDefinition nestedTypeDef in type.NestedTypes)
                foreach (TypeDefinition newNesterTypeDef in typeDef.
                    NestedTypes)
                    if (nestedTypeDef.Name.Equals(newNesterTypeDef.Name))
                        InsertType(nestedTypeDef, firstPass, newNesterTypeDef)
                            ;


            if (type.BaseType != null)
                // update the basetype
                if (type.BaseType.Scope == type.Scope)
                {
                    GlobalMapperEntry<TypeReference> basetypeEntry =
                        _globalMaps.TypeReferences.Lookup(type.BaseType);
                    if (basetypeEntry == null)
                        throw new Exceptions.IllegalOperationException("Unable
```

```
                    to locate the class '" + type.BaseType.FullName +
                    "' in the target assembly. This class is
                    inherited by the class '" + type.Name + "', which
                    you have specified should be inserted into the
                    target assembly. Please specify that the class '"
                    + type.BaseType.FullName + "' should be introduced
                    as well using the pointcut file.");
            else if (basetypeEntry.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable
                    to inherit the class '" + type.BaseType.FullName
                    + "' from class '" + type.Name + "', as '" + type.
                    BaseType.FullName + "' is inserted at multiple
                    locations in the target assembly (thus causing the
                    reference to this class to be ambiguous");
            else
                typeDef.BaseType = basetypeEntry.Reference;
        }
        else
        {
            if (!(YIIHAW.Weaver.Helper.IsAssemblyInRefs(typeDef.Module
                .AssemblyReferences, type.BaseType) || YIIHAW.Weaver.
                Helper.IsAssemblyTarget(type.BaseType, typeDef)))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not
                    possible to type check the usage of '" + type.
                    BaseType.FullName + "'. Please make sure that this
                    class is available from the target assembly.");

            if (YIIHAW.Weaver.Helper.IsAssemblyTarget(type.BaseType,
                typeDef))
                typeDef.BaseType = YIIHAW.Weaver.Helper.FindLocalType(
                    typeDef, type.BaseType);
            else
                typeDef.BaseType = typeDef.Module.Import(type.BaseType
                    );
        }


        // add interfaces
        foreach (TypeReference interfaceType in type.Interfaces)
        {
            if (interfaceType.Scope == type.Scope)
            {
                GlobalMapperEntry<TypeReference> interfaceEntry =
                    _globalMaps.TypeReferences.Lookup(interfaceType);
                if (interfaceEntry == null)
                    throw new Exceptions.IllegalOperationException("Unable
                        to locate the interface '" + interfaceType.Name +
                        "' in the target assembly. This interface is
                        implemented by the class '" + type.Name + "',
                        which you have specified should be inserted into
                        the target assembly. Please specify that the
                        interface '" + interfaceType.Name + "' should be
                        introduced as well using the pointcut file.");
                else if (interfaceEntry.IsAmbiguousReference)
                    throw new Exceptions.IllegalOperationException("Unable
                        to insert the type '" + type.Name + "', as the
                        interface '" + interfaceType.Name + "', which is
                        implemented by this type is inserted at multiple
                        locations in the target assembly (thus causing the
                        reference to this interface to be ambiguous");
                else
                    typeDef.Interfaces.Add(interfaceEntry.Reference);
```

```
                }
                else
                {
                    if (!(YIIHAW.Weaver.Helper.IsAssemblyInRefs(typeDef.Module
                        .AssemblyReferences, interfaceType) || YIIHAW.Weaver.
                        Helper.IsAssemblyTarget(interfaceType, typeDef)))
                        YIIHAW.Output.OutputFormatter.AddWarning("It is not
                            possible to type check the usage of '" +
                            interfaceType.FullName + "'. Please make sure that
                             this interface is available from the target
                            assembly.");

                    if (YIIHAW.Weaver.Helper.IsAssemblyTarget(interfaceType,
                        typeDef))
                        typeDef.Interfaces.Add(YIIHAW.Weaver.Helper.
                            FindLocalType(typeDef, interfaceType));
                    else
                        typeDef.Interfaces.Add(typeDef.Module.Import(
                            interfaceType));
                }

            }

            // add attributes
            foreach (CustomAttribute attribute in type.CustomAttributes)
                typeDef.CustomAttributes.Add(_weaver.CopyAndUpdateAttribute(
                    type, typeDef, _globalMaps, attribute));

        }
        foreach (MethodDefinition methodDef in type.Constructors) // add
            constructors
            InsertMethod(methodDef, typeDef, firstPass, false);

        foreach (MethodDefinition methodDef in type.Methods) // add methods
            InsertMethod(methodDef, typeDef, firstPass, false);

        foreach (FieldDefinition fieldDef in type.Fields) // add fields
            InsertField(fieldDef, typeDef, firstPass);

        foreach (PropertyDefinition propertyDef in type.Properties) // add
            properties
            InsertProperty(propertyDef, typeDef, firstPass);

        foreach (EventDefinition eventDef in type.Events) // add events
            InsertEvent(eventDef, typeDef, firstPass);
    }

    /// <summary>
    /// Finds a type in the aspect assembly, based on an insert statement.
    /// </summary>
    /// <param name="insert">The insert statement that tells which type to
        find.</param>
    /// <returns>The type found in the aspect assembly.</returns>
    private TypeDefinition FindType(Insert insert)
    {
        foreach (TypeDefinition type in _aspectAssembly.MainModule.Types)
        {
            // the replacement of '/' to '.' is done, as nested classes are
                written differently in the insert statement and in CeCil.
            if (!insert.Type.Equals(type.FullName.Replace('/', '.')))
                continue;
```

```
            if (Helper.CheckAccessSpecifier(type.Attributes, insert.Access))
                return type;

        }

        throw new Exceptions.ConstructNotFoundException("The class or
            interface '" + insert.Type + "' could not be found.");
    }

    /// <summary>
    /// Finds a field in the aspect assembly, based on an insert statement.
    /// </summary>
    /// <param name="insert">The insert statement that tells which field to
        find.</param>
    /// <returns>The field found in the aspect assembly.</returns>
    private FieldDefinition FindField(Insert insert)
    {
        foreach (TypeDefinition typeDef in _aspectAssembly.MainModule.Types)
        {
            if (typeDef.FullName.Equals(insert.Type)) // we have found the
                correct class - run through all fields in the class
            {
                foreach (FieldDefinition fieldDef in typeDef.Fields)
                {

                    // check the field name
                    if (!fieldDef.Name.Equals(insert.Name))
                        continue;

                    // check the access specifier
                    if (!(insert.InsertType == YIIHAW.Pointcut.InsertTypeEnum.
                        EVENT)) // event fields are always private so the
                            access specifier should not be checked for these
                            fields
                            if (!Helper.CheckAccessSpecifier(fieldDef.Attributes,
                                insert.Access))
                                continue;

                    // check the field type
                    if (!(insert.ReturnType.Type == YIIHAW.Pointcut.
                        ReturnTypeEnum.ANY) && !Helper.IsTypeEqual(insert.
                        ReturnType.SpecificType, fieldDef.FieldType))
                            continue;

                    // check static - instance
                    if (insert.InvocationKind == InvocationKindEnum.INSTANCE
                        && fieldDef.IsStatic)
                            continue;
                    if (insert.InvocationKind == InvocationKindEnum.STATIC &&
                        !fieldDef.IsStatic)
                            continue;

                    return fieldDef;
                }
            }

        }
        throw new Exceptions.ConstructNotFoundException("The field '" + insert
            .Name + "' could not be found in class " + insert.Type + ".");
    }

    /// <summary>
```

```
/// Finds a property in the aspect assembly, based on an insert statement.
/// </summary>
/// <param name="insert">The insert statement that tells which property to
     find.</param>
/// <returns>The property found in the aspect assembly.</returns>
private PropertyDefinition FindProperty(Insert insert)
{
    foreach (TypeDefinition typeDef in _aspectAssembly.MainModule.Types)
    {

        if (typeDef.FullName.Equals(insert.Type)) // we have found the
            correct class - run through all parameters in the class
            foreach (PropertyDefinition propertyDef in typeDef.Properties)
            {
                // Check the name
                if (!propertyDef.Name.Equals(insert.Name))
                    continue;

                if (propertyDef.GetMethod != null)
                {
                    // check the access specifier
                    if (!Helper.CheckAccessSpecifier(propertyDef.GetMethod
                        .Attributes, insert.Access))
                        continue;

                    // check the method type
                    if (!Helper.CheckInvocationKind(propertyDef.GetMethod,
                         insert.InvocationKind))
                        continue;
                }
                else
                {
                    //check the access specifier
                    if (!Helper.CheckAccessSpecifier(propertyDef.SetMethod
                        .Attributes, insert.Access))
                        continue;

                    //check the method type
                    if (!Helper.CheckInvocationKind(propertyDef.SetMethod,
                         insert.InvocationKind))
                        continue;
                }

                // check the property type
                if (insert.ReturnType.Type != ReturnTypeEnum.ANY && !
                    Helper.IsTypeEqual(insert.ReturnType.SpecificType,
                    propertyDef.PropertyType))
                    continue;

                return propertyDef;
            }
    }
    throw new Exceptions.ConstructNotFoundException("The property : " +
        insert.Name + " was not found in class : " + insert.Type + ".");
}


/// <summary>
/// Finds a method in the aspect assembly, based on an insert statement.
/// </summary>
/// <param name="insert">The insert statement that tells which method to
    find.</param>
```

```
/// <returns>The method found in the aspect assembly.</returns>
protected MethodDefinition FindMethod(Insert insert)
{
    foreach (TypeDefinition typeDef in _aspectAssembly.MainModule.Types)
    {

        if (typeDef.FullName.Equals(insert.Type)) // we have found the
            correct class - run through all enclosing methods
            foreach (MethodDefinition methodDef in typeDef.Methods) //
                check if this method matches the pointcut
            {
                // check the method name
                if (!methodDef.Name.Equals(insert.Method.Name))
                    continue;

                // check the access specifier
                if (!Helper.CheckAccessSpecifier(methodDef.Attributes,
                    insert.Access))
                    continue;

                // check the return type
                if (insert.ReturnType.Type == ReturnTypeEnum.VOID)
                {
                    if (!methodDef.ReturnType.ReturnType.FullName.Equals("
                        System.Void"))
                        continue;
                }
                else if (insert.ReturnType.Type != ReturnTypeEnum.ANY && !
                    Helper.IsTypeEqual(insert.ReturnType.SpecificType,
                    methodDef.ReturnType.ReturnType))
                    continue;

                // check the method type
                if (!Helper.CheckInvocationKind(methodDef, insert.
                    InvocationKind))
                    continue;

                // check the argument types
                if (!Helper.CheckArguments(insert.Method.ArgumentList,
                    methodDef.Parameters))
                    continue;

                return methodDef;
            }
    }
    throw new YIIHAW.Exceptions.ConstructNotFoundException("Specified
        method (" + insert.Method.Name + ") could not be found.");
}

/// <summary>
/// Inserts a method from the aspect into the target assembly.
/// </summary>
/// <param name="aspectMethod">The aspect method to insert.</param>
/// <param name="insert">The insert statement that tells where the method
    should be inserted.</param>
/// <param name="firstPass">A boolean indicating if this is the first pass
     of the insertion.</param>
/// <param name="addToOutput">A boolean indicating if the insertion of the
     method should be registered in the output.</param>
protected void InsertAspectMethod(MethodDefinition aspectMethod, Insert
    insert, bool firstPass, bool addToOutput)
{
```

```
        foreach (TypeDefinition typeDef in _targetAssembly.MainModule.Types)
        {
            // ignore all interfaces and modules
            if (typeDef.FullName.Equals("<Module>") || typeDef.IsInterface)
                continue;

            // ignore newly inserted types
            if (_globalMaps.Types.ContainsValue(typeDef))
                continue;

            // check that this target matches the type defined in the pointcut
                specification
            if (!Helper.CheckTarget(insert.TargetType, typeDef.FullName))
                continue;
            if (firstPass) // check if there is already a method with the same
                signature in the target type.
                foreach (MethodDefinition methodDef in typeDef.Methods)
                {
                    if (!methodDef.Name.Equals(aspectMethod.Name))
                        continue;

                    if (!(methodDef.IsStatic == aspectMethod.IsStatic))
                        continue;

                    if (!(methodDef.GenericParameters.Count == aspectMethod.
                        GenericParameters.Count))
                        continue;

                    if (!(methodDef.Parameters.Count == aspectMethod.
                        Parameters.Count))
                        continue;

                    bool argsEqual = true;
                    for (int i = 0; i < methodDef.Parameters.Count; i++)
                    {
                        if (!methodDef.Parameters[i].ParameterType.FullName.
                            Equals(aspectMethod.Parameters[i].ParameterType.
                            FullName))
                        {
                            argsEqual = false;
                            break;
                        }
                    }

                    if (argsEqual)
                        throw new Exceptions.IllegalOperationException("Method
                            '" + aspectMethod.Name + "' can not be inserted
                            into class '" + typeDef.FullName + "', as a method
                            with the same signature already exist in this
                            class.");
                }

            // insert the method into the target type.
            InsertMethod(aspectMethod, typeDef, firstPass, addToOutput);
        }
    }

    /// <summary>
    /// Inserts a method from the aspect assembly into a given target type.
    /// </summary>
    /// <param name="aspectMethod">The method to insert.</param>
    /// <param name="typeDef">The type to insert the message into</param>
```

```
/// <param name="firstPass">A boolean indicating if this is the first pass
     of the insertion.</param>
/// <param name="addToOutput">A boolean indicating if the insertion of the
     method should be registered in the output.</param>
private void InsertMethod(MethodDefinition aspectMethod, TypeDefinition
    typeDef, bool firstPass, bool addToOutput)
{
    if (firstPass) // this is the first pass - create a new method and
        store it in the mapper
    {
        MethodDefinition newMethod = new MethodDefinition(aspectMethod.
            Name, aspectMethod.Attributes, aspectMethod.ReturnType.
            ReturnType);
        _localMaps.Methods.Add(typeDef, aspectMethod, newMethod);
        _localMaps.MethodReferences.Add(typeDef, aspectMethod, newMethod);
        _globalMaps.Methods.Add(aspectMethod, newMethod);
        _globalMaps.MethodReferences.Add(aspectMethod, newMethod);
        if (aspectMethod.IsConstructor)
            typeDef.Constructors.Add(newMethod);
        else
            typeDef.Methods.Add(newMethod);
    }
    else // this is the second pass - instruct the weaver to insert the
        body of the method
    {
        MethodDefinition targetMethod = _localMaps.Methods.Lookup(typeDef,
             aspectMethod);
        if (targetMethod != null)
        {
            if (addToOutput)
                YIIHAW.Output.OutputFormatter.AddMethod(Helper.
                    MethodToString(aspectMethod), targetMethod.
                    DeclaringType.FullName);

            _weaver.InsertMethod(aspectMethod, targetMethod, _localMaps,
                _globalMaps);

            typeDef.Module.Import(typeDef);
        }
        else
            throw new Exceptions.InternalErrorException("Unable to
                retrieve method from mapper in second pass.");
    }
}

/// <summary>
/// Inserts a property from the aspect into the target assembly.
/// The setter and getter methods belonging to the property are also
    inserted.
/// </summary>
/// <param name="property">The property to insert.</param>
/// <param name="insert">The insert statement that tells where the
    property should be inserted.</param>
/// <param name="firstPass">A boolean indicating if this is the first pass
     of the insertion.</param>
private void InsertAspectProperty(PropertyDefinition property, Insert
    insert, bool firstPass)
{
    // insert the getter method
    if (property.GetMethod != null)
        InsertAspectMethod(property.GetMethod, insert, firstPass, false);
    // insert the setter method
```

```
        if (property.SetMethod != null)
            InsertAspectMethod(property.SetMethod, insert, firstPass, false);

        // find the types where the property should be inserted.
        foreach (TypeDefinition typeDef in _targetAssembly.MainModule.Types)
        {
            // ignore all interfaces and modules
            if (typeDef.FullName.Equals("<Module>") || typeDef.IsInterface)
                continue;

            // ignore newly inserted types
            if (_globalMaps.Types.ContainsValue(typeDef))
                continue;

            // check that this target matches the type defined in the pointcut
                specification
            if (!Helper.CheckTarget(insert.TargetType, typeDef.FullName))
                continue;

            InsertProperty(property, typeDef, firstPass);

            if (firstPass)
                YIIHAW.Output.OutputFormatter.AddProperty(property.
                    DeclaringType.FullName + "." + property.Name, typeDef.
                    FullName);
        }
    }

    /// <summary>
    /// Inserts a property into a given target.
    /// The methods belogning to the property should already be inserted, and
        registered in the local mapper.
    /// </summary>
    /// <param name="property">The property to insert.</param>
    /// <param name="typeDef">The type to insert the property into.</param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
        of the insertion.</param>
    private void InsertProperty(PropertyDefinition property, TypeDefinition
        typeDef, bool firstPass)
    {
        if (firstPass)
        {
            PropertyDefinition newProperty = property.Clone();
            newProperty.CustomAttributes.Clear(); //The attributes set by
                Clone() are not correct
            typeDef.Properties.Add(newProperty);
        }
        else
        {
            // find the property that was inserted in the first pass.
            foreach (PropertyDefinition newProperty in typeDef.Properties)
            {
                if (!newProperty.Name.Equals(property.Name))
                    continue;

                //set getter method to the inserted method
                if (property.GetMethod != null)
                    newProperty.GetMethod = _localMaps.Methods.Lookup(typeDef,
                        property.GetMethod);

                //set setter method to the inserted method
                if (property.SetMethod != null)
```

```
            newProperty.SetMethod = _localMaps.Methods.Lookup(typeDef,
                property.SetMethod);

        if (property.PropertyType.Scope == property.DeclaringType.
            Scope) // Trying to instantiate a type defined in the
            aspect assembly. This is only allowed if the reference is
            not ambiguous.
        {
            GlobalMapperEntry<TypeReference> mappedTypeEntry =
                _globalMaps.TypeReferences.Lookup(property.
                PropertyType);

            if (mappedTypeEntry == null)
                throw new Exceptions.IllegalOperationException("Unable
                    to insert the property '" + property.Name + "'
                    into '" + typeDef.FullName + "', as the property
                    type ('" + property.PropertyType.FullName + "') is
                    not defined in the target assembly. Please
                    specify that this type should be introduced using
                    the pointcut file.");
            else if (mappedTypeEntry.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable
                    to insert the property'" + property.Name + "'
                    into '" + typeDef.FullName + "', as the property
                    type is inserted at multiple locations (and is
                    thus ambiguous).");
            else
                newProperty.PropertyType = mappedTypeEntry.Reference;
        }
        else
        {
            // is the property type already available to the target
                assembly.
            if (!(YIIHAW.Weaver.Helper.IsAssemblyInRefs(typeDef.Module
                .AssemblyReferences, property.PropertyType) || YIIHAW.
                Weaver.Helper.IsAssemblyTarget(property.PropertyType,
                typeDef)))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not
                    possible to type check the usage of '" + property.
                    PropertyType.FullName + "'. Please make sure that
                    this class is available from the target assembly."
                    );

            // is the property type a type defined in the target, if
                so it does not need to be imported.
            if (YIIHAW.Weaver.Helper.IsAssemblyTarget(property.
                PropertyType, typeDef))
                newProperty.PropertyType = YIIHAW.Weaver.Helper.
                    FindLocalType(typeDef, property.PropertyType);
            else
                newProperty.PropertyType = typeDef.Module.Import(
                    property.PropertyType);
        }

        //insert attributes
        foreach (CustomAttribute attribute in property.
            CustomAttributes)
            newProperty.CustomAttributes.Add(_weaver.
                CopyAndUpdateAttribute(property.DeclaringType, typeDef
                , _globalMaps, attribute));

        // update the target assembly.
```

```
                typeDef.Module.Import(typeDef);
            }
        }
    }

    /// <summary>
    /// Inserts a field from the aspect into the target assembly.
    /// </summary>
    /// <param name="aspectField">The field to insert.</param>
    /// <param name="insert">The insert statement that tells where the field
        should be inserted.</param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
         of the insertion.</param>
    private void InsertAspectField(FieldDefinition aspectField, Insert insert,
         bool firstPass)
    {
        // find the types where the field should be inserted.
        foreach (TypeDefinition typeDef in _targetAssembly.MainModule.Types)
        {
            // ignore all interfaces and modules
            if (typeDef.FullName.Equals("<Module>") || typeDef.IsInterface)
                continue;

            // ignore newly inserted types
            if (_globalMaps.Types.ContainsValue(typeDef))
                continue;

            // check that this target matches the type defined in the pointcut
                 specification
            if (!Helper.CheckTarget(insert.TargetType, typeDef.FullName))
                continue;


            if (firstPass)
            {
                // check if there is already a field with the same name.
                foreach (FieldDefinition fieldDef in typeDef.Fields)
                    if (fieldDef.Name.Equals(aspectField.Name))
                        throw new Exceptions.IllegalOperationException("Field
                            '" + aspectField.Name + "' can not be inserted
                            into class '" + typeDef.FullName + "', as a field
                            with the same name already exist in this class.");

                YIIHAW.Output.OutputFormatter.AddField(aspectField.
                    DeclaringType.FullName + "." + aspectField.Name, typeDef.
                    FullName);
            }
            InsertField(aspectField, typeDef, firstPass);
        }
    }

    /// <summary>
    /// Inserts a field into a given target.
    /// </summary>
    /// <param name="aspectField">The field to insert.</param>
    /// <param name="target">The target type to insert the field into.</param>
    /// <param name="firstPass">A boolean indicating if this is the first pass
         of the insertion.</param>
    private void InsertField(FieldDefinition aspectField, TypeDefinition
        target, bool firstPass)
    {
        if (firstPass)
```

```
{
    // create a copy of the field, add it to the mappings and to the
        target type.
    FieldDefinition targetField = aspectField.Clone();
    _localMaps.Fields.Add(target, aspectField, targetField);
    _localMaps.FieldReferences.Add(target, aspectField, targetField);
    _globalMaps.Fields.Add(aspectField, targetField);
    _globalMaps.FieldReferences.Add(aspectField, targetField);

    targetField.CustomAttributes.Clear(); // The attributes that has
        been set by using Clone() are not correct

    target.Fields.Add(targetField);
    target.Module.Import(target);
}
else //this is the second pass
{
    // find the previous inserted field and update its type reference.
    FieldDefinition targetField = _localMaps.Fields.Lookup(target,
        aspectField);
    TypeReference typeRef = targetField.FieldType;
    if (typeRef.Scope == aspectField.DeclaringType.Scope) // Trying to
         instantiate a type defined in the aspect assembly. This is
        only allowed if the reference is not ambiguous.
    {
        GlobalMapperEntry<TypeReference> mappedTypeEntry = _globalMaps
            .TypeReferences.Lookup(typeRef);

        if (mappedTypeEntry == null)
            throw new Exceptions.IllegalOperationException("Unable to
                insert the field '" + aspectField.Name + "' into '" +
                target.FullName + "', as the field type ('" +
                aspectField.FieldType.FullName + "') is not defined in
                 the target assembly. Please specify that this type
                should be introduced using the pointcut file.");
        else if (mappedTypeEntry.IsAmbiguousReference)
            throw new Exceptions.IllegalOperationException("Unable to
                insert the field '" + aspectField.Name + "' into '" +
                target.FullName + "', as the field type is inserted at
                 multiple locations (and is thus ambiguous).");
        else
            targetField.FieldType = mappedTypeEntry.Reference;
    }
    else
    {
        if (!(YIIHAW.Weaver.Helper.IsAssemblyInRefs(target.Module.
            AssemblyReferences, typeRef) || YIIHAW.Weaver.Helper.
            IsAssemblyTarget(typeRef, target)))
            YIIHAW.Output.OutputFormatter.AddWarning("It is not
                possible to type check the usage of '" + aspectField.
                FieldType.FullName + "'. Please make sure that this
                class is available from the target assembly.");

        if (YIIHAW.Weaver.Helper.IsAssemblyTarget(targetField.
            FieldType, target))
            targetField.FieldType = YIIHAW.Weaver.Helper.FindLocalType
                (target, targetField.FieldType);
        else
            targetField.FieldType = target.Module.Import(aspectField.
                FieldType);
    }
```

```
                //insert attributes
                foreach (CustomAttribute attribute in aspectField.CustomAttributes
                    )
                    targetField.CustomAttributes.Add(_weaver.
                        CopyAndUpdateAttribute(aspectField.DeclaringType, target,
                        _globalMaps, attribute));

            }
        }
    }
}
```

InterceptHandler.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Mono.Cecil;
using Mono.Cecil.Cil;
using YIIHAW.Pointcut;
using YIIHAW.Weaver;
using YIIHAW.Exceptions;

namespace YIIHAW.Controller
{
    /// <summary>
    /// Handles the interception of methods in the target assembly, which are
        intercepted by methods from the aspect assemby.
    /// </summary>
    class InterceptHandler
    {

        private AssemblyDefinition _aspectAssembly;
        private AssemblyDefinition _targetAssembly;
        private LocalMapperCollection _localMaps;
        private GlobalMapperCollection _globalMaps;
        private Interception _weaver;

        /// <summary>
        /// Creates a new InterceptHandler and sets the data needed for the
            handler to function properly.
        /// </summary>
        /// <param name="aspectAssembly">The aspect assembly, where the
            interception methods are located.</param>
        /// <param name="targetAssembly">The target assembly which holds the
            methods that should be intercepted.</param>
        /// <param name="localMaps">A LocalMapperCollection which will be used
            throughout the interceptions.</param>
        /// <param name="globalMaps">A GlobalMapperCollection which will be used
            throughout the interceptions.</param>
        public InterceptHandler(AssemblyDefinition aspectAssembly,
            AssemblyDefinition targetAssembly, LocalMapperCollection localMaps,
            GlobalMapperCollection globalMaps)
        {
            _localMaps = localMaps;
            _globalMaps = globalMaps;
            _aspectAssembly = aspectAssembly;
            _targetAssembly = targetAssembly;
            _weaver = new Interception(localMaps, globalMaps);
        }
```

```
/// <summary>
/// Processes an around statement by finding the advice methods that are
    valid for the statement
/// and then intercept the methods in the target that are matching the
    pointcut in the statement.
/// </summary>
/// <param name="around">The pointcut represented as an around statement
    .</param>
public void ProcessStatement(Around around)
{
    List<MethodDefinition> adviceMethods = FindAdviceMethods(around.
        AdviceName, around.AdviceType);
    InterceptTargetMethods(adviceMethods, around);
}

/// <summary>
/// Finds the methods in target that matches the pointcut, and if there is
     an advice method
/// which can be used for intercepting the target method, it calls the
    weaver to get the two methods weaved together.
/// </summary>
/// <param name="adviceMethods">A list of advice methods that can be used
    for the interception.</param>
/// <param name="around">The around statement that tells which methods
    should be the target of interception.</param>
private void InterceptTargetMethods(List<MethodDefinition> adviceMethods,
    Around around)
{
    foreach (TypeDefinition typeDef in _targetAssembly.MainModule.Types)
    {
        if (!typeDef.IsInterface && Helper.CheckTarget(around.TargetType,
            typeDef.FullName)) // we have found the correct class - run
            through all enclosing methods
            foreach (MethodDefinition methodDef in typeDef.Methods) //
                check if this method matches the pointcut
            {
                // check if this method have been introduced previously
                if (_localMaps.Methods.ContainsValue(methodDef))
                    continue;

                // check the method name
                if (!Helper.CheckTarget(around.TargetMethod.Name,
                    methodDef.Name))
                    continue;

                // check the access specifier
                if (!Helper.CheckAccessSpecifier(methodDef.Attributes,
                    around.Access))
                    continue;

                // check the return type
                if (around.ReturnType.Type == ReturnTypeEnum.VOID)
                {
                    if (!methodDef.ReturnType.ReturnType.FullName.Equals("
                        System.Void"))
                        continue;
                }
                else if (around.ReturnType.Type != ReturnTypeEnum.ANY && !
                    Helper.IsTypeEqual(around.ReturnType.SpecificType,
                    methodDef.ReturnType.ReturnType))
                    continue;
```

```
                    // check the method type
                    if (!Helper.CheckInvocationKind(methodDef, around.
                        InvocationrType))
                        continue;

                    // check the argument types
                    if (!Helper.CheckArguments(around.TargetMethod.
                        ArgumentList, methodDef.Parameters))
                        continue;

                    // check the inherits property
                    if (!Helper.CheckInherit(around.Inherit, typeDef))
                        continue;

                    // check if there is an advice method that can be used for
                         the interception of the target method.
                    // if that is the case, get the weaver to weave the two
                        methods together.
                    MethodDefinition adviceMethod = FindBestMatch(methodDef,
                        adviceMethods);
                    if (adviceMethod != null)
                    {
                        YIIHAW.Output.OutputFormatter.AddMethodIntercepted(
                            Helper.MethodToString(methodDef), Helper.
                            MethodToString(adviceMethod));
                        _weaver.AroundIntercept(adviceMethod, methodDef);
                    }
                }
            }
        }

        /// <summary>
        /// Finds the best matching advice method to a given target method.
        /// Matching is on the signature of the method, where the return type has
            highest weight,
        /// and the parameter types of the advice must match the parameter types
            of the target method.
        /// It is okay if the advice takes lesser parameters than the target, as
            long as there is a match on all the parameter types of the advice.
        /// </summary>
        /// <param name="targetMethod">The target method to find a match for.</
            param>
        /// <param name="adviceMethods">A list of advice methods to find a match
            in.</param>
        /// <returns></returns>
        private MethodDefinition FindBestMatch(MethodDefinition targetMethod, List
            <MethodDefinition> adviceMethods)
        {
            // info about best match so far
            int numberOfParametersMatched = -1;
            bool returnTypeMatched = false;
            MethodDefinition bestMatch = null;

            // run through all available advice methods
            foreach (MethodDefinition advice in adviceMethods)
            {
                if (!advice.IsStatic && targetMethod.IsStatic) //If the advice
                    method is an instance method, the target method has to be an
                    instance method as well.
                    continue;

                TypeReference returnType = GetReturnType(advice); // get the
```

```
            return type of the current advice method

        if (returnType.FullName.Equals(targetMethod.ReturnType.ReturnType.
            FullName)) // the return type of the advice matches the return
             type of the target method
        {
            int paramThatMatch = GetParametersThatMatch(targetMethod,
                advice); // get the number of parameters that match the
                target method
            if (paramThatMatch > (returnTypeMatched ?
                numberOfParametersMatched : -1)) // this is the best match
                 so far
            {
                numberOfParametersMatched = paramThatMatch;
                bestMatch = advice;
                returnTypeMatched = true;
            }
        }
        else if(!returnTypeMatched && returnType is GenericParameter)
        // the return type of the advice does NOT match the return type of
             the target method
        // AND we have not yet found an advice method that matches the
            return type of the target method
        // AND the return type of the advice method is generic.
        {
            int paramThatMatch = GetParametersThatMatch(targetMethod,
                advice); // get the number of parameters that match the
                target method
            if (paramThatMatch > numberOfParametersMatched) // this is the
                 best match so far
            {
                numberOfParametersMatched = paramThatMatch;
                bestMatch = advice;
            }
        }
    }

    if (bestMatch != null)
        return bestMatch;
    else
    {
        YIIHAW.Output.OutputFormatter.AddMethodNotIntercepted(Helper.
            MethodToString(targetMethod));
        return null;
    }
}

/// <summary>
/// Counts how many parameters that matches between the target and advice
    methods.
/// If there is any parameters that does not match, the result is -1.
/// If the advice takes more parameters than the target the result is -1.
/// </summary>
/// <param name="targetMethod">The target method.</param>
/// <param name="adviceMethod">The advice method.</param>
/// <returns>An integer telling how many parameters that matched.</returns
    >
protected int GetParametersThatMatch(MethodDefinition targetMethod,
    MethodDefinition adviceMethod)
{
    if (targetMethod.Parameters.Count == 0 && adviceMethod.Parameters.
        Count == 0) // neither target method or advice method takes any
```

```
            parameters
            return 0;
        else if (adviceMethod.Parameters.Count > targetMethod.Parameters.Count
            ) // advice method takes more parameters than the target method -
            this is not a match
            return -1;
        else // target method takes a number of parameters - see how many of
            these parameters are matched by the advice method
        {
            int parametersMatched = 0; // parameters found so far

            // run through all arguments on the advice method
            foreach (ParameterDefinition paramDef in adviceMethod.Parameters)
            {
                if(paramDef.ParameterType.FullName.Equals(targetMethod.
                    Parameters[parametersMatched].ParameterType.FullName) &&
                    paramDef.ParameterType.Scope.Name.Equals(targetMethod.
                    Parameters[parametersMatched].ParameterType.Scope.Name))
                    // parameter match - increase the counter
                    parametersMatched++;
                else // parameter does not match - this advice method does not
                     match the target method
                    return -1;
            }

            return parametersMatched;
        }
    }

    /// <summary>
    /// Gets the return type of an advice method. The type might be generic,
        in which case a possible constraint on the type will define the return
         type.
    /// </summary>
    /// <param name="advice">The advice method, to get the return type from.</
        param>
    /// <returns>The return type of the given advice method.</returns>
    protected TypeReference GetReturnType(MethodDefinition advice)
    {

        if (advice.ReturnType.ReturnType is GenericParameter) // Returntype is
             generic -check constraints
        {
            GenericParameter genericParam = advice.ReturnType.ReturnType as
                GenericParameter;
            if (genericParam.Constraints.Count != 0) // there are constraints
                on the type - check if there is more than one (limitation by
                weaver).
            {
                if (genericParam.Constraints.Count != 1) // more than one
                    constraint is defined - this is illegal
                    throw new Exceptions.NotSupportedOperationException("It is
                         illegal to have more than one constraint on a generic
                         parameter (type '" + advice.DeclaringType.FullName +
                        "." + advice.Name + "').");
                else // only one constraint defined - return this constraint
                    return genericParam.Constraints[0];
            }
            else //no constraints - return the generic type
                return genericParam;
        }
        else //returntype is not generic - return the concrete type.
```

```
                return advice.ReturnType.ReturnType;
            }

            /// <summary>
            /// Finds the advice methods in the aspect assembly which matched the type
                 and name given.
            /// </summary>
            /// <param name="adviceName">The name that the methods should match.</
                param>
            /// <param name="adviceType">The type in which the methods should be
                located.</param>
            /// <returns></returns>
            protected List<MethodDefinition> FindAdviceMethods(string adviceName,
                string adviceType)
            {
                List<MethodDefinition> results = new List<MethodDefinition>();

                // run through all types defined in the aspect assembly
                foreach (TypeDefinition typeDef in _aspectAssembly.MainModule.Types)
                    if (typeDef.FullName.Equals(adviceType)) // we have found the
                        correct type
                        foreach (MethodDefinition methodDef in typeDef.Methods)
                            if (methodDef.Name.Equals(adviceName)) // we have found an
                                 advice method that matches the pointcut
                                results.Add(methodDef);

                return results;
            }
        }
    }
```

ModifyHandler.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAW.Pointcut;
using Mono.Cecil;
using YIIHAW.Weaver;

namespace YIIHAW.Controller
{
    /// <summary>
    /// Handles the modification of types in the target assembly.
    /// </summary>
    public class ModifyHandler
    {
        private AssemblyDefinition _aspectAssembly;
        private AssemblyDefinition _targetAssembly;
        private LocalMapperCollection _localMaps;
        private GlobalMapperCollection _globalMaps;

        /// <summary>
        /// Creates a new ModifyHandler and sets the data needed for the handler
            to function properly.
        /// </summary>
        /// <param name="aspectAssembly">The aspect assembly, where the
            modification types are located.</param>
        /// <param name="targetAssembly">The target assembly which holds the types
             that should be modified.</param>
        /// <param name="localMaps">A LocalMapperCollection which will be used
```

```
    throughout the modifications.</param>
/// <param name="globalMaps">A GlobalMapperCollection which will be used
    throughout the modifications.</param>
public ModifyHandler(AssemblyDefinition aspectAssembly, AssemblyDefinition
     targetAssembly, LocalMapperCollection localMaps,
    GlobalMapperCollection globalMaps)
{
    _aspectAssembly = aspectAssembly;
    _targetAssembly = targetAssembly;
    _localMaps = localMaps;
    _globalMaps = globalMaps;
}

/// <summary>
/// Processes an modify statement by finding the target types that matches
     the pointcut
/// and the do the modification to the found types.
/// </summary>
/// <param name="modify"></param>
public void ProcessStatement(Modify modify)
{
    List<TypeDefinition> targetTypes = FindTargetTypes(modify.TargetType);
    if (modify.ModifyType == ModifyTypeEnum.IMPLEMENT)
        ImplementInterface(targetTypes, modify);
    else
        SetNewBaseClass(targetTypes, modify);

}

/// <summary>
/// Changes the basetype of types given as parameter.
/// </summary>
/// <param name="targetTypes">A list of target types which should be
    modified.</param>
/// <param name="modify">The modify statement that tells what basetype to
    use in the modification.</param>
private void SetNewBaseClass(List<TypeDefinition> targetTypes, Modify
    modify)
{
    // find the aspect basetype
    TypeDefinition aspectBaseType = FindAspectType(modify, false);

    if (aspectBaseType == null) // no aspect basetype was found.
        throw new Exceptions.ConstructNotFoundException("The type '" +
            modify.InheritType + "' could not be found in the aspect
            assembly.");

    // as the new basetype must come from the aspect assembly, it should
        have been inserted into the target, and therefore be located in
        the global mapper.
    GlobalMapperEntry<TypeDefinition> mappedType = _globalMaps.Types.
        Lookup(aspectBaseType);
    if (mappedType == null)
        throw new Exceptions.ConstructNotFoundException("Unable to use the
             class '" + aspectBaseType.Name + "' as a basetype, as this
            class is not defined in the target assembly. If this class
            should be available in the target assembly, please specify
            that it should be inserted into the target assembly using the
            pointcut specification.");
    else if (mappedType.IsAmbiguousReference)
        throw new Exceptions.IllegalOperationException("Unable to use the
            class '" + aspectBaseType.Name + "'. The class is ambiguous,
```

```
                as it is inserted at multiple locations.");

        //check for each target type that it has the necessary methods and
            properties
        foreach (TypeDefinition type in targetTypes)
        {
            CheckTypeImplementation(type, aspectBaseType);

            // store a reference to the old basetype
            TypeReference oldBaseType = type.BaseType;

            // set the basetype
            type.BaseType = mappedType.Reference;

            // update all references in the methods.
            Modification modification = new Modification();

            foreach (MethodDefinition methodDef in type.Constructors)
                modification.ModifyMethod(methodDef, oldBaseType, mappedType.
                    Reference);

            foreach (MethodDefinition methodDef in type.Methods)
                modification.ModifyMethod(methodDef, oldBaseType, mappedType.
                    Reference);

            // update the target assembly
            type.Module.Import(type);
        }
    }

    /// <summary>
    /// Makes the target types implement an interface.
    /// </summary>
    /// <param name="targetTypes">A list of target types, which should be
        modified to implement the interface.</param>
    /// <param name="modify">The modify statement that tells which interface
        to use in the modification.</param>
    private void ImplementInterface(List<TypeDefinition> targetTypes, Modify
        modify)
    {
        //find the aspect interface
        TypeDefinition aspectInterface = FindAspectType(modify, true);

        if (aspectInterface == null) // no aspect interface found.
            throw new Exceptions.ConstructNotFoundException("The interface '"
                + modify.InheritType + "' could not be found in the aspect
                assembly.");

        // as the new interface must come from the aspect assembly, it should
            have been inserted into the target, and therefore be located in
            the global mapper.
        GlobalMapperEntry<TypeDefinition> mappedType = _globalMaps.Types.
            Lookup(aspectInterface);
        if (mappedType == null)
            throw new Exceptions.ConstructNotFoundException("Unable to
                implement interface '" + aspectInterface.Name + "', as this
                interface is not defined in the target assembly. If this
                interface should be available in the target assembly, please
                specify that it should be inserted into the target assembly
                using the pointcut specification.");
        else if (mappedType.IsAmbiguousReference)
            throw new Exceptions.IllegalOperationException("Unable to
```

```
                implement interface '" + aspectInterface.Name + "'. The
                interface is ambiguous, as it is inserted at multiple
                locations.");


        //check for each targettype that it has the necessary methods and
            properties
        foreach (TypeDefinition type in targetTypes)
        {
            CheckTypeImplementation(type, aspectInterface);

            //insert the interface.
            type.Interfaces.Add(mappedType.Reference);
        }
    }

    /// <summary>
    /// Finds a type in the aspect assembly, based on the pointcut.
    /// </summary>
    /// <param name="modify">The pointcut represented as a modify statement.</
        param>
    /// <param name="isInterface">A boolean indicating if it is an interface
        that should be found.</param>
    /// <returns>The type found in the aspect assembly that matched the
        pointcut.</returns>
    private TypeDefinition FindAspectType(Modify modify,bool isInterface)
    {
        // go through each type.
        foreach (TypeDefinition typeDef in _aspectAssembly.MainModule.Types)
        {
            // check whether the actual type is an interface, and whether it
                is an interface that should be found.
            if ((!typeDef.IsInterface && isInterface) || (typeDef.IsInterface
                && !isInterface))
                continue;

            // check the name
            if (typeDef.FullName.Equals(modify.InheritType))
                return typeDef;
        }
        return null;
    }

    /// <summary>
    /// Checks if a given type can implement/extend a given aspect type.
    /// </summary>
    /// <param name="type">The type that should be checked.</param>
    /// <param name="aspectType">The aspect type to check against.</param>
    private void CheckTypeImplementation(TypeDefinition type, TypeDefinition
        aspectType)
    {
        // check all methods in the aspect type.
        foreach (MethodDefinition aspectMethodDef in aspectType.Methods)
        {
            // only abstract methods need to be implemented in the type.
            if (!aspectMethodDef.IsAbstract)
                continue;

            bool methodFound = false;
            foreach (MethodDefinition methodDef in type.Methods)
            {
                // check the method name
```

```
            if (!aspectMethodDef.Name.Equals(methodDef.Name))
                continue;

            // check the access specifier
            if(!Helper.CheckAccessSpecifier(methodDef.Attributes,
                aspectMethodDef.Attributes))
                continue;

            // check the return type
            if (!methodDef.ReturnType.ReturnType.Name.Equals(
                aspectMethodDef.ReturnType.ReturnType.Name))
                continue;

            // check the invocation kind
            if (!(aspectMethodDef.IsStatic == methodDef.IsStatic))
                continue;

            // check the argument types
            if (!Helper.CheckArguments(aspectMethodDef.Parameters,
                methodDef.Parameters))
                continue;

            // check generic parameters
            if (!Helper.CheckGenericParameters(aspectMethodDef.
                GenericParameters, methodDef.GenericParameters))
                continue;

            methodFound = true;
        }

        if (!methodFound)
            throw new Exceptions.ConstructNotFoundException("The class '"
                + type.FullName + "' does not contain abstract method '" +
                 aspectMethodDef.Name + "'. Please specify that this
                method should be inserted into this class using the
                pointcut file.");
    }
    // check all properties in the aspect type.
    foreach (PropertyDefinition aspectProperty in aspectType.Properties)
    {
        bool propertyFound = false;
        foreach (PropertyDefinition targetProperty in type.Properties)
        {
            // check the property name
            if (!aspectProperty.Name.Equals(targetProperty.Name))
                continue;

            // check the property type
            if (!aspectProperty.PropertyType.FullName.Equals(
                targetProperty.PropertyType.FullName))
                continue;

            propertyFound = true;
        }

        if (!propertyFound)
            throw new Exceptions.ConstructNotFoundException("The class '"
                + type.FullName + "' does not contain interface property '
                " + aspectProperty.Name + "'. Please specify that this
                property should be inserted into this class using the
                pointcut file.");
    }
```

```
        }

        /// <summary>
        /// Finds all the types in the target assembly who's full name matches a
            given string.
        /// The string can be from a pointput, and therefore used *'s.
        /// </summary>
        /// <param name="targetType">The name to match against.</param>
        /// <returns>A list of types in the target assembly that matched.</returns
            >
        private List<TypeDefinition> FindTargetTypes(string targetType)
        {

            List<TypeDefinition> types = new List<TypeDefinition>();

            foreach (TypeDefinition type in _targetAssembly.MainModule.Types)
                if (Helper.CheckTarget(targetType, type.FullName))
                    types.Add(type);

            return types;
        }
    }
}
```

Helper.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Mono.Cecil;
using Mono.Cecil.Cil;
using YIIHAW.Pointcut;

namespace YIIHAW.Controller
{
    /// <summary>
    /// Contains helper methods for the InsertHandler, InterceptHandler and
        ModifyHandler classes.
    /// </summary>
    public static class Helper
    {
        /// <summary>
        /// Checks if the name of a target matches the given pointcut.
        /// </summary>
        /// <param name="pointcut">The pointcut value.</param>
        /// <param name="target">The target value.</param>
        /// <returns>A boolean indicating if the target matches the pointcut.</
            returns>
        public static bool CheckTarget(string pointcut, string target)
        {
            if (pointcut.Equals("*") && !target.Contains("/")) // matches all
                types (except nested types)
                return true;
            else if (pointcut.StartsWith("*") && !target.Contains("/")) // matches
                any type followed by a specific type (*.something) (except nested
                types)
                return (target.EndsWith(pointcut.Substring(1)));
            else if (pointcut.EndsWith("*") && !target.Contains("/")) // matches a
                specific type followed by any type (something.*) (except nested
                types)
                return (target.StartsWith(pointcut.Substring(0, pointcut.Length -
```

```
                1)));
        else // matches only fully qualified types
            return target.Equals(pointcut);
    }

    /// <summary>
    /// Checks if the access specfier of a method matches the given pointcut.
    /// </summary>
    /// <param name="attributes">The MethodAttributes of the method to check
        .</param>
    /// <param name="target">The pointcut to check against.</param>
    /// <returns>A boolean indicating if the methods access specifier matches
        the pointcut.</returns>
    public static bool CheckAccessSpecifier(MethodAttributes attributes,
        AccessEnum target)
    {
        if (target == AccessEnum.INTERNAL && !((attributes & MethodAttributes.
            Assem) == MethodAttributes.Assem))
            return false;
        else if (target == AccessEnum.PRIVATE && !((attributes &
            MethodAttributes.Private) == MethodAttributes.Private))
            return false;
        else if (target == AccessEnum.PROTECTED && !((attributes &
            MethodAttributes.Family) == MethodAttributes.Family))
            return false;
        else if (target == AccessEnum.PUBLIC && !((attributes &
            MethodAttributes.Public) == MethodAttributes.Public))
            return false;

        return true;
    }

    /// <summary>
    /// Checks if two methods has the same access specifier.
    /// </summary>
    /// <param name="targetAttributes">The first method.</param>
    /// <param name="aspectAttributes">The second method.</param>
    /// <returns>A boolean indicating if the two methods had the same access
        specifier or not.</returns>
    public static bool CheckAccessSpecifier(MethodAttributes targetAttributes,
         MethodAttributes aspectAttributes)
    {
        if (((targetAttributes & MethodAttributes.Assem) == MethodAttributes.
            Assem) && !((aspectAttributes & MethodAttributes.Assem) ==
            MethodAttributes.Assem))
            return false;
        else if (((targetAttributes & MethodAttributes.Private) ==
            MethodAttributes.Private) && !((aspectAttributes &
            MethodAttributes.Private) == MethodAttributes.Private))
            return false;
        else if (((targetAttributes & MethodAttributes.Family) ==
            MethodAttributes.Family) && !((aspectAttributes & MethodAttributes
            .Family) == MethodAttributes.Family))
            return false;
        else if (((targetAttributes & MethodAttributes.Public) ==
            MethodAttributes.Public) && !((aspectAttributes & MethodAttributes
            .Public) == MethodAttributes.Public))
            return false;

        return true;
    }
```

```
/// <summary>
/// Checks if the access specfier of a type matches the given pointcut.
/// </summary>
/// <param name="attributes">The TypeAttributes of the type to check.</
    param>
/// <param name="target">The pointcut to check against.</param>
/// <returns>A boolean indicating if the types access specifier matches
    the pointcut.</returns>
public static bool CheckAccessSpecifier(TypeAttributes attributes,
    AccessEnum target)
{
    if (target == AccessEnum.PRIVATE && !(attributes == TypeAttributes.
        NotPublic))
        return false;

    else if (target == AccessEnum.PUBLIC && !((attributes & TypeAttributes
        .Public) == TypeAttributes.Public))
        return false;

    return true;
}

/// <summary>
/// Checks if the access specfier of a field matches the given pointcut.
/// </summary>
/// <param name="attributes">The FieldAttributes of the type to check.</
    param>
/// <param name="target">The pointcut to check against.</param>
/// <returns>A boolean indicating if the fields access specifier matches
    the pointcut.</returns>
public static bool CheckAccessSpecifier(FieldAttributes attributes,
    AccessEnum target)
{
    if (target == AccessEnum.INTERNAL && !((attributes & FieldAttributes.
        Assembly) == FieldAttributes.Assembly))
        return false;
    else if (target == AccessEnum.PRIVATE && !((attributes &
        FieldAttributes.Private) == FieldAttributes.Private))
        return false;
    else if (target == AccessEnum.PROTECTED && !((attributes &
        FieldAttributes.Family) == FieldAttributes.Family))
        return false;
    else if (target == AccessEnum.PUBLIC && !((attributes &
        FieldAttributes.Public) == FieldAttributes.Public))
        return false;

    return true;
}

/// <summary>
/// Checks if the invocation kind of a method matches the given pointcut.
/// </summary>
/// <param name="target">The method to check.</param>
/// <param name="memberType">The pointcut to check against.</param>
/// <returns>A boolean indicating if the methods invocation kind matches
    the pointcut.</returns>
public static bool CheckInvocationKind(MethodDefinition target,
    InvocationKindEnum memberType)
{
    if (memberType == InvocationKindEnum.INSTANCE && target.IsStatic)
        return false;
    else if (memberType == InvocationKindEnum.STATIC && !target.IsStatic)
```

```
            return false;

        return true;
    }

    /// <summary>
    /// Checks if the arguments of a method matches the given pointcut.
    /// </summary>
    /// <param name="target">The pointcut to check against.</param>
    /// <param name="parameters">The arguments of the method.</param>
    /// <returns></returns>
    public static bool CheckArguments(ArgumentList target,
        ParameterDefinitionCollection parameters)
    {
        if (target.ArgumentType == ArgTypeEnum.NONE)
        {
            if (parameters.Count > 0) //There are parameters but there should
                be none.
                return false;
        }
        else if (target.ArgumentType == ArgTypeEnum.ANY)
            return true;
        else
        {
            if (target.Arguments.Count != parameters.Count)
                return false;

            int i = 0;
            foreach (string arg in target.Arguments)
            {
                if (!IsTypeEqual(arg, parameters[i].ParameterType))
                    return false;

                i++;
            }
        }

        return true;
    }

    /// <summary>
    /// Checks if the arguments of an aspect method matches the arguments of a
         target method.
    /// </summary>
    /// <param name="aspect">The aspect methods arguments.</param>
    /// <param name="target">The target methods arguments.</param>
    /// <returns>A boolean indicating if the arguments of the two methods
        matches.</returns>
    public static bool CheckArguments(ParameterDefinitionCollection aspect,
        ParameterDefinitionCollection target)
    {
        if (target.Count != aspect.Count)
            return false;

        for (int i = 0; i < aspect.Count; i++)
            if (!aspect[i].Name.Equals(target[i].Name))
                return false;

        return true;
    }

    /// <summary>
```

```
/// Checks if a type name matches a given type.
/// For the primitive types defined in the .Net framework, the short form
    versions are handled.
/// </summary>
/// <param name="type">The type name to check.</param>
/// <param name="typeReference">The type to check against.</param>
/// <returns>A boolean indicating if the type name matches the type.</
    returns>
public static bool IsTypeEqual(string type, TypeReference typeReference)
{
    if (type.Equals(typeReference.Name))
        return true;
    if (type.Equals(typeReference.FullName))
        return true;
    if (type.Equals("byte"))
        return typeReference.FullName.Equals("System.Byte");
    if (type.Equals("sbyte"))
        return typeReference.FullName.Equals("System.SByte");
    if (type.Equals("int"))
        return typeReference.FullName.Equals("System.Int32");
    if (type.Equals("uint"))
        return typeReference.FullName.Equals("System.UInt32");
    if (type.Equals("short"))
        return typeReference.FullName.Equals("System.Int16");
    if (type.Equals("ushort"))
        return typeReference.FullName.Equals("System.UInt16");
    if (type.Equals("long"))
        return typeReference.FullName.Equals("System.Int64");
    if (type.Equals("ulong"))
        return typeReference.FullName.Equals("System.UInt64");
    if (type.Equals("float"))
        return typeReference.FullName.Equals("System.Single");
    if (type.Equals("double"))
        return typeReference.FullName.Equals("System.Double");
    if (type.Equals("char"))
        return typeReference.FullName.Equals("System.Char");
    if (type.Equals("bool"))
        return typeReference.FullName.Equals("System.Boolean");
    if (type.Equals("object"))
        return typeReference.FullName.Equals("System.Object");
    if (type.Equals("string"))
        return typeReference.FullName.Equals("System.String");
    if (type.Equals("decimal"))
        return typeReference.FullName.Equals("System.Decimal");

    return false;
}

/// <summary>
/// Formats a methods signature as a string.
/// </summary>
/// <param name="method">The method.</param>
/// <returns>A string representing the signature of the given method.</
    returns>
public static string MethodToString(MethodDefinition method)
{
    StringBuilder sb = new StringBuilder();
    sb.Append(YIIHAW.Output.OutputFormatter.GetTypeShortFormat(method.
        ReturnType.ReturnType.FullName));
    sb.Append(" ");
    sb.Append(method.DeclaringType.FullName);
    sb.Append(":");
```

```
        sb.Append(method.Name);
        sb.Append("(");

        bool firstParam = true;
        foreach (ParameterDefinition paramDef in method.Parameters)
        {
            if (!firstParam)
                sb.Append(", ");

            firstParam = false;
            sb.Append(YIIHAW.Output.OutputFormatter.GetTypeShortFormat(
                paramDef.ParameterType.FullName));
        }
        sb.Append(")");

        return sb.ToString();
    }

    /// <summary>
    /// Checks if a type inherits a given type (non recursive).
    /// </summary>
    /// <param name="inherit">The basetype.</param>
    /// <param name="typeDef">The type to check.</param>
    /// <returns>A boolean indicating if the type inherits the specified type
        .</returns>
    internal static bool CheckInherit(Inherit inherit, TypeDefinition typeDef)
    {
        if (inherit.InheritType == InheritTypeEnum.ANY) //Inherit type is
            irrelevant as all types are matched.
            return true;

        if (typeDef.BaseType.FullName.Equals(inherit.SpecificType)) //Basetype
             matches.
            return true;


        foreach (TypeReference typeRef in typeDef.Interfaces) //Check the
            implemented interfaces.
            if (typeRef.FullName.Equals(inherit.SpecificType))
                return true;

        return false;
    }

    /// <summary>
    /// Checks if the generic parameters of two methods are identical.
    /// </summary>
    /// <param name="aspect">Generic parameters from the aspect method.</param
        >
    /// <param name="target">Generic parameters from the target method.</param
        >
    /// <returns>A boolean indicating if the parameters are identical.</
        returns>
    internal static bool CheckGenericParameters(GenericParameterCollection
        aspect, GenericParameterCollection target)
    {
        if (target.Count != aspect.Count)
            return false;

        //The only thing necessary to check, are the constraints on the types.
        for (int i = 0; i < aspect.Count; i++)
            if (!aspect[i].Constraints.Equals(target[i].Constraints))
```

```
                return false;

            return true;
        }
    }
}
```

Mediator.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using YIIHAW.Pointcut;
using Mono.Cecil;
using System.IO;
using YIIHAW.Weaver;
using YIIHAW.Output;

namespace YIIHAW.Controller
{
    /// <summary>
    /// The Mediator is the main controller of the program, and it is also the
        starting point of the program.
    /// </summary>
    public class Mediator
    {
        /// <summary>
        /// The method that starts the weaver.
        /// </summary>
        /// <param name="args">The arguments for the weaver. <pointcut> <target> <
            aspect> [out] [-v]</param>
        public static void Main(string[] args)
        {
            OutputFormatter.Reset();
            // check to see that the right number og arguments are specified.
            if (args.Length < 3 || args.Length > 5)
                Console.WriteLine("Usage: yiihaw <pointcut> <target> <aspect> [out
                    ] [-v]");
            else
            {
                bool verbose = false; //indicates if the output should be in
                    verbose mode or not.
                if ((args.Length == 4 && args[3].Equals("-v")) || (args.Length ==
                    5 && args[4].Equals("-v")))
                    verbose = true;

                AssemblyDefinition targetAssembly;
                AssemblyDefinition aspectAssembly;
                LocalMapperCollection localMaps = new LocalMapperCollection();
                GlobalMapperCollection globalMaps = new GlobalMapperCollection();

                // open the target assembly.
                try
                {
                    targetAssembly = AssemblyFactory.GetAssembly(args[1]);
                }
                catch (FileNotFoundException)
                {
                    OutputFormatter.AddException("Target assembly could not be
                        found");
                    OutputFormatter.PrintOutput(verbose);
```

```
        Console.ReadLine();
        return;
    }
    catch (Exception e)
    {
        OutputFormatter.AddException("An error occurred while reading
            target assembly: " + e.Message);
        OutputFormatter.PrintOutput(verbose);
        Console.ReadLine();
        return;
    }

    // open the aspect assembly.
    try
    {
        aspectAssembly = AssemblyFactory.GetAssembly(args[2]);
    }
    catch (FileNotFoundException)
    {
        OutputFormatter.AddException("Aspect assembly could not be
            found");
        OutputFormatter.PrintOutput(verbose);
        Console.ReadLine();
        return;
    }
    catch (Exception e)
    {
        OutputFormatter.AddException("An error occurred while reading
            aspect assembly: " + e.Message);
        OutputFormatter.PrintOutput(verbose);
        Console.ReadLine();
        return;
    }

    // parse the pointcut file.
    Parser parser;
    try
    {
        parser = new Parser(new YIIHAW.Pointcut.LexicalAnalysis.
            Scanner(args[0]));
        parser.Parse();
    }
    catch (YIIHAW.Pointcut.Exceptions.InputFileNotFoundException)
    {
        OutputFormatter.AddException("Pointcut file could not be found
            ");
        OutputFormatter.PrintOutput(verbose);
        Console.ReadLine();
        return;
    }
    catch (YIIHAW.Pointcut.Exceptions.ParseError e)
    {
        OutputFormatter.AddException("Error while parsing pointcut
            file: " + e.Message);
        OutputFormatter.PrintOutput(verbose);
        Console.ReadLine();
        return;
    }

    // take action of the statements created by the parser, based on
        the pointcut file.
    try
```

```
{
    // insertion first.
    InsertHandler insertHandler = new InsertHandler(aspectAssembly
        , targetAssembly, localMaps, globalMaps);

    // do first pass
    foreach (Insert insert in parser.InsertStatements)
        insertHandler.ProcessStatement(insert, true);

    // do second pass
    foreach (Insert insert in parser.InsertStatements)
        insertHandler.ProcessStatement(insert, false);

    // modification second.
    ModifyHandler modifyHandler = new ModifyHandler(aspectAssembly
        , targetAssembly, localMaps, globalMaps);

    foreach (Modify modify in parser.ModifyStatements)
        modifyHandler.ProcessStatement(modify);

    // and interception last.
    InterceptHandler interceptHandler = new InterceptHandler(
        aspectAssembly, targetAssembly, localMaps, globalMaps);

    foreach (Around around in parser.AroundStatements)
        interceptHandler.ProcessStatement(around);

    // if there is an output file specified make sure it got a
        useable extention, and change the name registered as
        metadata in the new assembly.
    if (args.Length >= 4 && args[3] != "-v")
    {
        if (args[3].Substring(args[3].Length - 4) != ".exe" &&
            args[3].Substring(args[3].Length - 4) != ".dll")
        { // there is no extention on the output filename, use the
             extention from the target assembly.
            args[3] += targetAssembly.MainModule.Name.Substring(
                targetAssembly.MainModule.Name.Length - 4);
        }
        // change the name in the metadata of the outputted
            assembly, so that it matched the name of the outputted
             file.
        int lastslash = args[3].LastIndexOf('\\');
        if (lastslash == -1)
            lastslash = args[3].LastIndexOf('/');
        if (lastslash == -1)
        { // there is no path specification in the output filename
            . Just use the name, without the extention.
            targetAssembly.MainModule.Name = args[3];
            targetAssembly.Name.Name = args[3].Substring(0, args
                [3].Length - 4);
        }
        else
        { // there is a path specification in the output filename,
             use only the filename part of the string.
            targetAssembly.MainModule.Name = args[3].Substring(
                lastslash + 1);
            targetAssembly.Name.Name = args[3].Substring(lastslash
                +1, args[3].Length -lastslash -5);
        }
        // save the weaved assembly with the new filename.
        AssemblyFactory.SaveAssembly(targetAssembly, args[3]);
```

```
                    }
                    else
                        // overwrite the target assembly, with the weaved assembly
                            .
                        AssemblyFactory.SaveAssembly(targetAssembly, args[1]);
                }
                catch (YIIHAW.Exceptions.InternalErrorException e)
                {
                    OutputFormatter.AddInternalException(e.Message);
                }
                catch (Exception e)
                {
                    OutputFormatter.AddException(e.Message);
                }
                finally
                {
                    OutputFormatter.PrintOutput(verbose);
                    Console.WriteLine("Press <Enter> to end the weaving");
                    Console.ReadLine();
                }
            }
        }
    }
}
```

Appendix X

Source code for YIIHAW - Weaver

Introduction.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Mono.Cecil;
using Mono.Cecil.Cil;

namespace YIIHAW.Weaver
{
    /// <summary>
    /// Used for the weaving part of an introductions.
    /// </summary>
    public class Introduction
    {
        /// <summary>
        /// Inserts a given aspect method into a given target method,
        /// so that the target becomes a copy of the aspect method.
        /// </summary>
        /// <param name="aspect">The aspect method to insert.</param>
        /// <param name="target">The target method to weave the aspect into.</
            param>
        /// <param name="localMaps">A LocalMapperCollection which will be used
            throughout the introduction of the method.</param>
        /// <param name="globalMaps">A GlobalMapperCollection which will be used
            throughout the introduction of the method.</param>
        public void InsertMethod(MethodDefinition aspect, MethodDefinition target,
             LocalMapperCollection localMaps, GlobalMapperCollection globalMaps)
        {
            if (!aspect.IsAbstract)
            {
                // insert locals
                foreach (VariableDefinition varDef in aspect.Body.Variables)
                {
                    // special action is needed if the variable is an arrayType,
                        as the actual type is then stored at another place.
                    if (varDef.VariableType is ArrayType)
                    {
                        HandleArrayVar(varDef, aspect, target, globalMaps);
                    }
                    else if (varDef.VariableType.Scope == aspect.DeclaringType.
                        Scope) // Trying to instantiate a type defined in the
                        aspect assembly. This is only allowed if the reference is
                        not ambiguous.
                    {
```

```
                GlobalMapperEntry<TypeReference> mappedTypeEntry =
                    globalMaps.TypeReferences.Lookup(varDef.VariableType);

                if (mappedTypeEntry == null)
                    throw new Exceptions.IllegalOperationException("Unable
                        to access the type '" + varDef.VariableType.
                        FullName + "' from '" + target.DeclaringType.
                        FullName + "." + target.Name + "', as '" + varDef.
                        VariableType.FullName + "' is not defined in the
                        target assembly. Please specify that this type
                        should be introduced using the pointcut file.");
                else if (mappedTypeEntry.IsAmbiguousReference)
                    throw new Exceptions.IllegalOperationException("Unable
                        to access the type '" + varDef.VariableType.
                        FullName + "' from '" + target.DeclaringType.
                        FullName + "." + target.Name + "'. The reference
                        is ambiguous, as the type is inserted at multiple
                        locations.");
                else
                    varDef.VariableType = mappedTypeEntry.Reference;
            }
            else
            {
                if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                    AssemblyReferences, varDef.VariableType) || Helper.
                    IsAssemblyTarget(varDef.VariableType, target.
                    DeclaringType)))
                    YIIHAW.Output.OutputFormatter.AddWarning("It is not
                        possible to type check the instantiation of '" +
                        varDef.VariableType.FullName + "'. Please make
                        sure that this class is available from the target
                        assembly.");

                if (Helper.IsAssemblyTarget(varDef.VariableType, target.
                    DeclaringType))
                    varDef.VariableType = Helper.FindLocalType(target.
                        DeclaringType, varDef.VariableType);
                else
                    varDef.VariableType = target.DeclaringType.Module.
                        Import(varDef.VariableType);
            }

            target.Body.Variables.Add(varDef);
        }

        // insert init locals metadata
        target.Body.InitLocals = aspect.Body.InitLocals;
    }

    // insert arguments
    InsertMethodArguments(aspect, target, globalMaps);

    // insert generic parameters
    InsertGenericParameters(aspect, target, globalMaps);

    // update returntype
    UpdateReturnType(aspect, target, globalMaps);

    target.Attributes = aspect.Attributes;

    //insert attributes
    InsertAttributes(aspect, target, globalMaps);
```

```
if (!aspect.IsAbstract)
{
    // run through all instructions in the aspect body and check if
        special action is needed.
    foreach (Instruction curInstr in aspect.Body.Instructions)
    {
        CilWorker worker = aspect.Body.CilWorker;

        // check if this instruction is a "call"
        if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
            MethodCallOpCodesArray))
            HandleMethodReference(curInstr, target, aspect, worker,
                localMaps, globalMaps);
        // check if this opcode is a reference to a field (store or
            load)
        else if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
            LoadFieldOpCodesArray) || Helper.IsOpcodeEqual(curInstr.
            OpCode, Helper.StoreFieldOpCodesArray))
            HandleFieldAccess(curInstr, aspect, target, worker,
                localMaps, globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Newobj))
            HandleNewobj(curInstr, target, aspect, worker, globalMaps)
                ;

        else if (curInstr.OpCode.Equals(OpCodes.Newarr))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Box))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);

        else if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
            UnBoxOpCodesArray))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Castclass))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Isinst))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Constrained))
            HandleConstrained(curInstr, target, aspect, worker,
                globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Ldobj))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Stobj))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);

        else if (curInstr.OpCode.Equals(OpCodes.Cpobj))
            HandleTypeImport(curInstr, target, aspect, worker,
                globalMaps);
```

```
                else if (curInstr.OpCode.Equals(OpCodes.Mkrefany))
                    HandleTypeImport(curInstr, target, aspect, worker,
                        globalMaps);

                else if (curInstr.OpCode.Equals(OpCodes.Refanytype))
                    HandleTypeImport(curInstr, target, aspect, worker,
                        globalMaps);

                else if (curInstr.OpCode.Equals(OpCodes.Refanyval))
                    HandleTypeImport(curInstr, target, aspect, worker,
                        globalMaps);

                else if (curInstr.OpCode.Equals(OpCodes.Sizeof))
                    HandleTypeImport(curInstr, target, aspect, worker,
                        globalMaps);

                else if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
                    ElementLoadAndStoreWithTokenArray))
                    HandleTypeImport(curInstr, target, aspect, worker,
                        globalMaps);

                else if (curInstr.OpCode.Equals(OpCodes.Ldtoken))
                {
                    if (curInstr.Operand is MethodReference)
                        HandleMethodReference(curInstr, target, aspect, worker
                            , localMaps, globalMaps);
                    else if (curInstr.Operand is TypeReference)
                        HandleTypeImport(curInstr, target, aspect, worker,
                            globalMaps);
                    else
                        HandleFieldAccess(curInstr, aspect, target, worker,
                            localMaps, globalMaps);
                }

                else // no changes are needed for this instruction
                    target.Body.CilWorker.Append(curInstr);

            }
        }
    }

    /// <summary>
    /// Handles the case where a variable that should be inserted is of an
        arraytype.
    /// </summary>
    /// <param name="varDef">The variable that has the arraytype.</param>
    /// <param name="aspect">The advice method.</param>
    /// <param name="target">The target method.</param>
    /// <param name="globalMaps">The global mappings to make lookups in.</
        param>
    private void HandleArrayVar(VariableDefinition varDef, MethodDefinition
        aspect, MethodDefinition target, GlobalMapperCollection globalMaps)
    {
        TypeReference typeRef = (varDef.VariableType as ArrayType).ElementType
            ;

        if (typeRef.Scope == aspect.DeclaringType.Scope) // Trying to
            instantiate a type defined in the aspect assembly. This is only
            allowed if the reference is not ambiguous.
        {
            GlobalMapperEntry<TypeReference> mappedTypeEntry = globalMaps.
```

```
                TypeReferences.Lookup(typeRef);

            if (mappedTypeEntry == null)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the type '" + typeRef.FullName + "' from '" +
                    target.DeclaringType.FullName + "." + target.Name + "', as
                     '" + typeRef.FullName + "' is not defined in the target
                    assembly. Please specify that this type should be
                    introduced using the pointcut file.");
            else if (mappedTypeEntry.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the type '" + typeRef.FullName + "' from '" +
                    target.DeclaringType.FullName + "." + target.Name + "'.
                    The reference is ambiguous, as the type is inserted at
                    multiple locations.");
            else
                (varDef.VariableType as ArrayType).ElementType =
                    mappedTypeEntry.Reference;
        }
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, typeRef) || Helper.IsAssemblyTarget(
                typeRef, target.DeclaringType)))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                    to type check the instantiation of '" + typeRef.FullName +
                     "'. Please make sure that this class is available from
                    the target assembly.");

            if (Helper.IsAssemblyTarget(typeRef, target.DeclaringType))
                (varDef.VariableType as ArrayType).ElementType = Helper.
                    FindLocalType(target.DeclaringType, typeRef);
            else
                (varDef.VariableType as ArrayType).ElementType = target.
                    DeclaringType.Module.Import(typeRef);
        }
    }

    /// <summary>
    /// Sets the same attributes on the target method as the ones set on the
        aspect method.
    /// </summary>
    /// <param name="aspect">The aspect method.</param>
    /// <param name="target">The target method to set the attributes on.</
        param>
    /// <param name="globalMaps">The global mappings to make lookups in.</
        param>
    private void InsertAttributes(MethodDefinition aspect, MethodDefinition
        target, GlobalMapperCollection globalMaps)
    {
        foreach (CustomAttribute attribute in aspect.CustomAttributes)
        {
            // a new copy must be made for each attribute.
            CustomAttribute newAttribute = CopyAndUpdateAttribute(aspect.
                DeclaringType, target.DeclaringType, globalMaps, attribute);

            target.CustomAttributes.Add(newAttribute);
        }
    }

    /// <summary>
    /// Creates a copy of a given attribute, and updates the references
```

```
/// of the copy so that it works in the target assembly.
/// </summary>
/// <param name="aspect">The aspect method.</param>
/// <param name="target">The target method.</param>
/// <param name="globalMaps">The global mappings to make lookups in.</
    param>
/// <param name="attribute">The attribute to make a copy of.</param>
/// <returns></returns>
public CustomAttribute CopyAndUpdateAttribute(TypeReference aspect,
    TypeReference target, GlobalMapperCollection globalMaps,
    CustomAttribute attribute)
{
    // copy the attribute
    CustomAttribute newAttribute = attribute.Clone();
    // update the type of the attribute reference.
    TypeReference typeRef = attribute.Constructor.DeclaringType;

    if (typeRef.Scope == aspect.Scope) // Trying to use a type defined in
        the aspect assembly. This is only allowed if the reference is not
        ambiguous.
    {
        GlobalMapperEntry<TypeReference> mappedTypeEntry = globalMaps.
            TypeReferences.Lookup(typeRef);

        if (mappedTypeEntry == null)
            throw new Exceptions.IllegalOperationException("Unable to
                access the attribute '" + typeRef.FullName + "' from '" +
                target.FullName + "', as '" + typeRef.FullName + "' is not
                 defined in the target assembly. Please specify that this
                type should be introduced using the pointcut file.");
        else if (mappedTypeEntry.IsAmbiguousReference)
            throw new Exceptions.IllegalOperationException("Unable to
                access the attribute '" + typeRef.FullName + "' from '" +
                target.FullName + "'. The reference is ambiguous, as the
                type is inserted at multiple locations.");
        else
        {
            newAttribute.Constructor.DeclaringType = mappedTypeEntry.
                Reference;
            newAttribute.Constructor = Helper.FindLocalMethod(target,
                newAttribute.Constructor);
        }
    }
    else
    {
        if (!(Helper.IsAssemblyInRefs(target.Module.AssemblyReferences,
            typeRef) || Helper.IsAssemblyTarget(typeRef, target)))
            YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                to type check the usage of '" + typeRef.FullName + "'.
                Please make sure that this class is available from the
                target assembly.");

        if (Helper.IsAssemblyTarget(typeRef, target))
            newAttribute.Constructor = Helper.FindLocalMethod(target,
                newAttribute.Constructor);
        else
            newAttribute.Constructor = target.Module.Import(newAttribute.
                Constructor);
    }
    return newAttribute;
}
```

```
/// <summary>
/// Handles instructions being inserted that is referencing a method.
/// </summary>
/// <param name="curInstr">The instruction which is referencing a method
    .</param>
/// <param name="target">The target method where the instruction should be
     inserted.</param>
/// <param name="aspect">The aspect method from where the instruction
    originates.</param>
/// <param name="worker">A CilWorker.</param>
/// <param name="localMpas">The local mappings to make lookups in.</param>
/// <param name="globalMaps">The global mappings to make lookups in.</
    param>
private void HandleMethodReference(Instruction curInstr, MethodDefinition
    target, MethodDefinition aspect, CilWorker worker,
    LocalMapperCollection localMaps , GlobalMapperCollection globalMaps)
{
    MethodReference methodRef = curInstr.Operand as MethodReference;

    // check that none of the generic parameters defined on the advice
        class are passed as argument to other method - this is not allowed
         as the generic parameters do not exist at runtime (and would thus
         fail)
    foreach (GenericParameter genericParameter in methodRef.
        GenericParameters)
        if (aspect.DeclaringType.GenericParameters.Contains(
            genericParameter))
            throw new Exceptions.IllegalOperationException("The generics
                parameters defined on the advice class can not be used
                when calling methods ('" + aspect.DeclaringType.FullName +
                 "." + aspect.Name + "').");

    if (methodRef.DeclaringType == aspect.DeclaringType) // the call is to
         a method defined in the same class as the aspect method - map the
         method call to the method that was created in the target type
        during the first pass
    {
        MethodReference mappedMethod = localMaps.MethodReferences.Lookup(
            target.DeclaringType, methodRef);
        if (mappedMethod != null) // the method was found in the mapper -
            insert a call to this method
        methodRef = mappedMethod;
        else // method was not found in the mapper - throw an exception
            throw new Exceptions.ConstructNotFoundException("Unable to
                invoke method '" + methodRef.Name + "' from '" + target.
                Name + "', as '" + methodRef.Name + "' is not defined in
                the target assembly. If this method should be available in
                 the target assembly, please specify that this method
                should be inserted into the target assembly using the
                pointcut specification.");
    }
    // Trying to use a method defined in the aspect assembly. This is only
         allowed if the reference is not ambiguous.
    else if (methodRef.DeclaringType.Scope == aspect.DeclaringType.Scope)
    {
        GlobalMapperEntry<MethodReference> mappedMethodRef = globalMaps.
            MethodReferences.Lookup(methodRef);
        if (mappedMethodRef == null)
            throw new Exceptions.ConstructNotFoundException("Unable to
                access the method '" + methodRef.Name + "' from '" +
                target.Name + "', as '" + methodRef.Name + "' is not
                defined in the target assembly. If this method should be
```

```
                available in the target assembly, please specify that it
                should be inserted into the target assembly using the
                pointcut file.");
            else if (mappedMethodRef.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the method '" + methodRef.Name + "' from '" +
                    target.Name + "'. The reference is ambiguous, as the
                    method is inserted at multiple locations.");
            else
                methodRef = mappedMethodRef.Reference;
        }
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, methodRef.DeclaringType) || Helper.
                IsAssemblyTarget(methodRef.DeclaringType, target.DeclaringType
                )))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                    to type check the use of '" + methodRef.DeclaringType.
                    FullName + "'. Please make sure that this class is
                    available from the target assembly.");

            if (Helper.IsAssemblyTarget(methodRef.DeclaringType, target.
                DeclaringType))
                methodRef = Helper.FindLocalMethod(target.DeclaringType,
                    methodRef);
            else
                methodRef = target.DeclaringType.Module.Import(methodRef);

        }
        target.Body.CilWorker.Append(worker.Create(curInstr.OpCode,methodRef))
            ;
    }

    /// <summary>
    /// Handles the .constrained instruction.
    /// </summary>
    /// <param name="curInstr">The .constrained instruction.</param>
    /// <param name="target">The target method.</param>
    /// <param name="aspect">The advice method.</param>
    /// <param name="worker">A CilWorker.</param>
    /// <param name="globalMaps">The global mappings to make lookups in.</
        param>
    private void HandleConstrained(Instruction curInstr, MethodDefinition
        target, MethodDefinition aspect, CilWorker worker,
        GlobalMapperCollection globalMaps)
    {
        TypeReference typeRef = curInstr.Operand as TypeReference;
        // if it is not a generic type, just handle it as any other
            instruction with a typeReference operand.
        if (!(typeRef is GenericParameter))
        {
            HandleTypeImport(curInstr, target, aspect, worker, globalMaps);
            return;
        }

        target.Body.CilWorker.Append(curInstr);
    }

    /// <summary>
    /// Inserts the generic parameters from an aspect method into a target
        method.
```

```
/// </summary>
/// <param name="aspect">The aspect method.</param>
/// <param name="target">The target method to insert the parameters into
    .</param>
/// <param name="globalMaps">The global mappings to make lookups in.</
    param>
private static void InsertGenericParameters(MethodDefinition aspect,
    MethodDefinition target, GlobalMapperCollection globalMaps)
{
    foreach (GenericParameter genericParam in aspect.GenericParameters)
    {
        // make a copy of the parameter
        GenericParameter newGenericParam = new GenericParameter(
            genericParam.Name, target);
        // add the constraints
        foreach (TypeReference typeRef in genericParam.Constraints)
        {
            if (typeRef.Scope == aspect.DeclaringType.Scope) // Trying to
                instantiate a type defined in the aspect assembly. This is
                 only allowed if the reference is not ambiguous.
            {
                GlobalMapperEntry<TypeReference> mappedTypeEntry =
                    globalMaps.TypeReferences.Lookup(typeRef);

                if (mappedTypeEntry == null)
                    throw new Exceptions.IllegalOperationException("Unable
                         to access the type '" + typeRef.FullName + "'
                        from '" + target.DeclaringType.FullName + "." +
                        target.Name + "', as '" + typeRef.FullName + "' is
                         not defined in the target assembly. Please
                        specify that this type should be introduced using
                        the pointcut file.");
                else if (mappedTypeEntry.IsAmbiguousReference)
                    throw new Exceptions.IllegalOperationException("Unable
                         to access the type '" + typeRef.FullName + "'
                        from '" + target.DeclaringType.FullName + "." +
                        target.Name + "'. The reference is ambiguous, as
                        the type is inserted at multiple locations.");
                else
                    newGenericParam.Constraints.Add(mappedTypeEntry.
                        Reference);
            }
            else
            {
                if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                    AssemblyReferences, typeRef) || Helper.
                    IsAssemblyTarget(typeRef, target.DeclaringType)))
                    YIIHAW.Output.OutputFormatter.AddWarning("It is not
                        possible to type check the usage of '" + typeRef.
                        FullName + "'. Please make sure that this class is
                         available from the target assembly.");

                if (Helper.IsAssemblyTarget(typeRef, target.DeclaringType)
                    )
                    newGenericParam.Constraints.Add(Helper.FindLocalType(
                        target.DeclaringType, typeRef));
                else
                    newGenericParam.Constraints.Add(target.DeclaringType.
                        Module.Import(typeRef));
            }
        }
```

```
            target.GenericParameters.Add(newGenericParam);
        }
    }

    /// <summary>
    /// Inserts the argument types from a aspect method into a target method.
    /// </summary>
    /// <param name="aspect">The aspect method.</param>
    /// <param name="target">The target method.</param>
    /// <param name="globalMaps">The global mappings to make lookups in.</
        param>
    private static void InsertMethodArguments(MethodDefinition aspect,
        MethodDefinition target, GlobalMapperCollection globalMaps)
    {
        foreach (ParameterDefinition paramDef in aspect.Parameters)
        {
            // create copy of the parameter
            ParameterDefinition newParamDef = new ParameterDefinition(paramDef
                .Name, paramDef.Sequence, paramDef.Attributes, paramDef.
                ParameterType);

            if (paramDef.ParameterType.Scope == aspect.DeclaringType.Scope) //
                Trying to use a type defined in the aspect assembly. This is
                only allowed if the reference is not ambiguous.
            {
                GlobalMapperEntry<TypeReference> mappedTypeEntry = globalMaps.
                    TypeReferences.Lookup(paramDef.ParameterType);

                if (mappedTypeEntry == null)
                    throw new Exceptions.IllegalOperationException("Unable to
                        access the type '" + paramDef.ParameterType.FullName +
                        "' from '" + target.DeclaringType.FullName + "." +
                        target.Name + "', as '" + paramDef.ParameterType + "'
                        is not defined in the target assembly. Please specify
                        that this type should be introduced using the pointcut
                        file.");
                else if (mappedTypeEntry.IsAmbiguousReference)
                    throw new Exceptions.IllegalOperationException("Unable to
                        access the type '" + paramDef.ParameterType.FullName +
                        "' from '" + target.DeclaringType.FullName + "." +
                        target.Name + "'. The reference is ambiguous, as the
                        type is inserted at multiple locations.");
                else
                    newParamDef.ParameterType = mappedTypeEntry.Reference;
            }
            else
            {
                if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                    AssemblyReferences, aspect.ReturnType.ReturnType) ||
                    Helper.IsAssemblyTarget(aspect.ReturnType.ReturnType,
                    target.DeclaringType)))
                    YIIHAW.Output.OutputFormatter.AddWarning("It is not
                        possible to type check the usage of '" + paramDef.
                        ParameterType + "'. Please make sure that this class
                        is available from the target assembly.");

                if (Helper.IsAssemblyTarget(paramDef.ParameterType, target.
                    DeclaringType))
                    newParamDef.ParameterType = Helper.FindLocalType(target.
                        DeclaringType, paramDef.ParameterType);
                else
                    newParamDef.ParameterType = target.DeclaringType.Module.
```

```
                    Import(paramDef.ParameterType);
            }

            target.Parameters.Add(newParamDef);
        }
    }

    /// <summary>
    /// Update the return type of the target method to that of the aspect
        method.
    /// </summary>
    /// <param name="aspect">The aspect method.</param>
    /// <param name="target">The target method.</param>
    /// <param name="globalMaps">The global mappings to make lookups in.</
        param>
    private static void UpdateReturnType(MethodDefinition aspect,
        MethodDefinition target, GlobalMapperCollection globalMaps)
    {
        if (aspect.ReturnType.ReturnType.Scope == aspect.DeclaringType.Scope)
            // Trying to instantiate a type defined in the aspect assembly.
            This is only allowed if the reference is not ambiguous.
        {
            GlobalMapperEntry<TypeReference> mappedTypeEntry = globalMaps.
                TypeReferences.Lookup(aspect.ReturnType.ReturnType);

            if (mappedTypeEntry == null)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the type '" + aspect.ReturnType.ReturnType.FullName
                     + "' from '" + target.DeclaringType.FullName + "." +
                    target.Name + "', as '" + aspect.ReturnType.ReturnType.
                    FullName + "' is not defined in the target assembly.
                    Please specify that this type should be introduced using
                    the pointcut file.");
            else if (mappedTypeEntry.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the type '" + aspect.ReturnType.ReturnType.FullName
                     + "' from '" + target.DeclaringType.FullName + "." +
                    target.Name + "'. The reference is ambiguous, as the type
                    is inserted at multiple locations.");
            else
                target.ReturnType.ReturnType = mappedTypeEntry.Reference;
        }
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, aspect.ReturnType.ReturnType) || Helper.
                IsAssemblyTarget(aspect.ReturnType.ReturnType, target.
                DeclaringType)))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                    to type check the usage of '" + aspect.ReturnType.
                    ReturnType.FullName + "'. Please make sure that this class
                     is available from the target assembly.");

            if (Helper.IsAssemblyTarget(aspect.ReturnType.ReturnType, target.
                DeclaringType))
                target.ReturnType.ReturnType = Helper.FindLocalType(target.
                    DeclaringType, aspect.ReturnType.ReturnType);
            else
                target.ReturnType.ReturnType = target.DeclaringType.Module.
                    Import(aspect.ReturnType.ReturnType);
        }
    }
```

```
/// <summary>
/// Given an instruction that has a TypeReference as operand, this type is
     imported into the target.
/// The instruction should come from an advice.
/// </summary>
/// <param name="curInstr">An instruction which has a TypeReference as
    operand.</param>
/// <param name="target">The target to insert it into.</param>
/// <param name="aspect">The advice from where the instruction originates
    .</param>
/// <param name="worker">A CilWorker.</param>
/// <param name="globalMaps">The global mappings to make lookups in.</
    param>
private void HandleTypeImport(Instruction curInstr, MethodDefinition
    target, MethodDefinition aspect, CilWorker worker,
    GlobalMapperCollection globalMaps)
{
    TypeReference typeRef = curInstr.Operand as TypeReference;
    GlobalMapperEntry<TypeReference> typeRefEntry = globalMaps.
        TypeReferences.Lookup(typeRef);

    // Trying to instantiate a type defined in the aspect assembly. This
        is only allowed if the reference is not ambiguous.
    if (typeRef.Scope == aspect.DeclaringType.Scope)
        if (typeRefEntry == null)
            throw new Exceptions.IllegalOperationException("Unable to
                instantiate '" + typeRef.FullName + "' from '" + target.
                DeclaringType.FullName + "." + target.Name + "', as '" +
                typeRef.FullName + "' is not defined in the target
                assembly. Please specify that this type should be
                introduced using the pointcut file.");
        else if (typeRefEntry.IsAmbiguousReference)
            throw new Exceptions.IllegalOperationException("Unable to
                instantiate '" + typeRef.FullName + "' from '" + target.
                DeclaringType.FullName + "." + target.Name + "'. The
                reference is ambiguous, as the type is inserted at
                multiple locations.");
        else
            typeRef = typeRefEntry.Reference;
    else
    {
        if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
            AssemblyReferences, typeRef) || Helper.IsAssemblyTarget(
            typeRef, target.DeclaringType)))
            YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                to type check the instantiation of '" + typeRef.
                DeclaringType.FullName + "'. Please make sure that this
                class is available from the target assembly.");

        if (Helper.IsAssemblyTarget(typeRef, target.DeclaringType))
            typeRef = Helper.FindLocalType(target.DeclaringType, typeRef);
        else
            typeRef = target.DeclaringType.Module.Import(typeRef);
    }


    target.Body.CilWorker.Append(worker.Create(curInstr.OpCode,typeRef));
}
```

```
/// <summary>
/// Takes a newobj instruction from the advice, and makes sure that it
    will work in the target.
/// </summary>
/// <param name="curInstr">The newobj instruction.</param>
/// <param name="target">The target to insert the instruction into.</param
    >
/// <param name="aspect">The advice from where the instruction originates
    .</param>
/// <param name="worker">A CilWorker.</param>
/// <param name="globalMaps">The global mappings to make lookups in.</
    param>
private void HandleNewobj(Instruction curInstr, MethodDefinition target,
    MethodDefinition aspect, CilWorker worker, GlobalMapperCollection
    globalMaps)
{
    MethodReference methodRef = curInstr.Operand as MethodReference;
    GlobalMapperEntry<MethodReference> methodRefEntry = globalMaps.
        MethodReferences.Lookup(methodRef);

    // Trying to instantiate a type defined in the aspect assembly. This
        is only allowed if the reference is not ambiguous.
    if (methodRef.DeclaringType.Scope == aspect.DeclaringType.Scope)
        if (methodRefEntry == null)
            throw new Exceptions.IllegalOperationException("Unable to
                instantiate '" + methodRef.DeclaringType.FullName + "'
                from '" + target.DeclaringType.FullName + "." + target.
                Name + "', as '" + methodRef.DeclaringType.FullName + "'
                is not defined in the target assembly. Please specify that
                 this type should be introduced using the pointcut file.")
                ;
        else if (methodRefEntry.IsAmbiguousReference)
            throw new Exceptions.IllegalOperationException("Unable to
                instantiate '" + methodRef.DeclaringType.FullName + "'
                from '" + target.DeclaringType.FullName + "." + target.
                Name + "'. The reference is ambiguous, as the type is
                inserted at multiple locations.");
        else
            methodRef = methodRefEntry.Reference;

    else
    {
        if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
            AssemblyReferences, methodRef.DeclaringType) || Helper.
            IsAssemblyTarget(methodRef.DeclaringType, target.DeclaringType
            )))
            YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                to type check the instantiation of '" + methodRef.
                DeclaringType.FullName + "'. Please make sure that this
                class is available from the target assembly.");

        if (Helper.IsAssemblyTarget(methodRef.DeclaringType, target.
            DeclaringType))
            methodRef = Helper.FindLocalMethod(target.DeclaringType,
                methodRef);
        else
            methodRef = target.DeclaringType.Module.Import(methodRef);
    }

    target.Body.CilWorker.Append(worker.Create(curInstr.OpCode,methodRef))
        ;
}
```

```
/// <summary>
/// Handles instructions which has an operand of type FieldReference.
/// The field reference is update, so that it works in the target assembly
    .
/// </summary>
/// <param name="curInstr">The instruction which has the FieldReference as
     operand.</param>
/// <param name="aspect">The target method.</param>
/// <param name="target">The advice method.</param>
/// <param name="worker">A CilWorker.</param>
/// <param name="localMaps">The local mappings to make lookups in.</param>
/// <param name="globalMaps">The global mappings to make lookups in.</
    param>
private void HandleFieldAccess(Instruction curInstr, MethodDefinition
    aspect, MethodDefinition target, CilWorker worker,
    LocalMapperCollection localMaps, GlobalMapperCollection globalMaps)
{
    Instruction newInstr;
    FieldReference fieldRef = curInstr.Operand as FieldReference;

    if (fieldRef.DeclaringType == aspect.DeclaringType) // the opcode
        refers to a field defined in the same class as the aspect method -
         map the reference to the field that was created in the target
        type during the first pass
    {
        FieldReference mappedField = localMaps.FieldReferences.Lookup(
            target.DeclaringType, fieldRef);
        if (mappedField != null) // the field was found in the mapper -
            insert a reference to this field
            newInstr = worker.Create(curInstr.OpCode, mappedField);
        else // field was not found in the mapper - throw an exception
            throw new Exceptions.ConstructNotFoundException("Unable to
                access field '" + fieldRef.Name + "' from '" + target.Name
                + "', as '" + fieldRef.Name + "' is not defined in the
                target assembly. If this field should be available in the
                target assembly, please specify that this field should be
                inserted into the target assembly using the pointcut
                specification.");
    }
    else if (fieldRef.DeclaringType.Scope == aspect.DeclaringType.Scope)
        // this is a reference to a field inside the aspect assembly, but
        outside the declaring type of the aspect method currently being
        inserted - check if the reference is ambiguous
    {
        GlobalMapperEntry<FieldReference> mappedField = globalMaps.
            FieldReferences.Lookup(fieldRef);
        if (mappedField == null)
            throw new Exceptions.ConstructNotFoundException("Unable to
                access field '" + fieldRef.Name + "' from '" + target.Name
                + "', as '" + fieldRef.Name + "' is not defined in the
                target assembly. If this field should be available in the
                target assembly, please specify that this field should be
                inserted into the target assembly using the pointcut
                specification.");
        else if (mappedField.IsAmbiguousReference)
            throw new Exceptions.IllegalOperationException("Unable to
                access field '" + fieldRef.Name + "' from '" + target.Name
                + "'. The reference is ambiguous, as the field is
                inserted at multiple locations.");
        else
            newInstr = worker.Create(curInstr.OpCode, mappedField.
```

```
                    Reference);
            }
            else // this is a reference to some field located outside the aspect
                assembly - add this field reference to the target method and
                import the declaring type of the field
            {
                FieldReference newFieldRef = fieldRef;

                if (Helper.IsAssemblyTarget(fieldRef.DeclaringType, target.
                    DeclaringType))
                    newFieldRef = Helper.FindLocalField(target, fieldRef);
                else
                    newFieldRef = target.DeclaringType.Module.Import(fieldRef);
                newInstr = worker.Create(curInstr.OpCode, newFieldRef);
            }

            target.Body.CilWorker.Append(newInstr);
        }
    }
}
```

Interception.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Mono.Cecil;
using Mono.Cecil.Cil;

namespace YIIHAW.Weaver
{
    /// <summary>
    /// Used for the weaving part of an interception.
    /// </summary>
    public class Interception
    {
        protected RecursiveDictionary<Instruction> _instructionMapping; // local
            mapping used when an instruction is replaced with another instruction
        private LocalMapperCollection _localMaps;
        private GlobalMapperCollection _globalMaps;
        private bool _proceedInvoked = false; //indicating if the advice has had a
             call to proceed.

        /// <summary>
        /// Creates a new Interception object and sets the data needed for the
            object to function properly.
        /// </summary>
        /// <param name="localMaps">A LocalMapperCollection which will be used
            throughout the interceptions.</param>
        /// <param name="globalMaps">A GlobalMapperCollection which will be used
            throughout the interceptions.</param>
        public Interception(LocalMapperCollection localMaps,
            GlobalMapperCollection globalMaps)
        {
            _localMaps = localMaps;
            _globalMaps = globalMaps;
            _instructionMapping = new RecursiveDictionary<Instruction>();
        }

        /// <summary>
        /// This is the entrance point from outside. Given an advice and a target
```

```
        the method
    /// makes an interception of the target method with the advice method.
    /// </summary>
    /// <param name="advice">The advice to use in the interception.</param>
    /// <param name="target">The target of the interception.</param>
    public void AroundIntercept(MethodDefinition advice, MethodDefinition
        target)
    {
        _proceedInvoked = false;
        _instructionMapping.Clear(); // clear all previous instructions stored
        MethodBody originalAdviceBody = advice.Clone().Body;

        //The clone method sets all Next and Previous reference to null. These
            references need to be updated manually.
        for (int index = 0; index < originalAdviceBody.Instructions.Count - 1;
            index++)
        {
            originalAdviceBody.Instructions[index].Next = originalAdviceBody.
                Instructions[index + 1];
            originalAdviceBody.Instructions[index + 1].Previous =
                originalAdviceBody.Instructions[index];
        }

        //The clone method sets the index to 0 for all local variables. These
            indexes need to be updated manually.
        for (int index = 0; index < originalAdviceBody.Variables.Count; index
            ++)
            originalAdviceBody.Variables[index].Index = index;

        //The clone method sets the methods declaring type to null
        originalAdviceBody.Method.DeclaringType = advice.DeclaringType;

        CilWorker worker = target.Body.CilWorker; // get a cil worker that can
            be used to access the original target body
        target.Body = new MethodBody(target); // clear the target body

        // update the target body so that it is ready to be inserted whenever
            we reach a call to Proceed()
        UpdateTargetBody(worker, target.Body, advice.Body, target.
            DeclaringType);

        // copy the body of the advice to the target, and take action on the
            special instructions.
        HandleAdvice(target, advice, worker);

        // post processing of the weaved body.
        UpdateWeavedTargetBody(worker, target.Body);

        //Insert exception and finally handlers.
        AddHandlers(worker.GetBody().ExceptionHandlers, target.Body);
        AddHandlers(advice.Body.ExceptionHandlers, target.Body);


        //Set the advice body back to its original form.
        advice.Body = originalAdviceBody;
    }

    /// <summary>
    /// Updates the references to local variables to the right one,
    /// and optimezes the opcode for load and store of local variables.
    /// </summary>
    /// <param name="worker">The worker of the original target body.</param>
```

```
/// <param name="targetBody">The MethodBody to update.</param>
protected void UpdateWeavedTargetBody(CilWorker worker, MethodBody
    targetBody)
{
    // remove local variables if necessary
    if (!_proceedInvoked) // the Proceed() method has not been invoked -
        remove all local variables that were copied from the original body
        , as these are no longer needed
        foreach (VariableDefinition varDef in worker.GetBody().Variables)
            targetBody.Variables.Remove(varDef);

    // all local variables are updated to have the correct Index parameter
        .
    for (int index = 0; index < targetBody.Variables.Count; index++)
        targetBody.Variables[index].Index = index;

    for (int index = 0; index < targetBody.Instructions.Count; index++)
    {
        Instruction instr = targetBody.Instructions[index];

        // access to localvariables (loading) needs to be updated with a
            new index
        if (Helper.IsOpcodeEqual(instr.OpCode, Helper.
            LoadLocalOpCodesArray) && !Helper.IsOpcodeEqual(instr.OpCode,
            Helper.LoadAdressOpCodesArray))
        {
            int number = (instr.Operand as VariableDefinition).Index;

            // the instrution to use instead of the orginal "load"
                instruction
            Instruction newInstr;
            switch (number)
            {
                case 0:
                    newInstr = targetBody.CilWorker.Create(OpCodes.Ldloc_0
                        );
                    break;
                case 1:
                    newInstr = targetBody.CilWorker.Create(OpCodes.Ldloc_1
                        );
                    break;
                case 2:
                    newInstr = targetBody.CilWorker.Create(OpCodes.Ldloc_2
                        );
                    break;
                case 3:
                    newInstr = targetBody.CilWorker.Create(OpCodes.Ldloc_3
                        );
                    break;
                default:
                    if (number < 256)
                        newInstr = targetBody.CilWorker.Create(OpCodes.
                            Ldloc_S, targetBody.Variables[number]);
                    else
                        newInstr = targetBody.CilWorker.Create(OpCodes.
                            Ldloc, targetBody.Variables[number]);
                    break;
            }

            // the replacement of an instruction needs to be registered
            _instructionMapping.Add(instr, newInstr);
            targetBody.Instructions[index] = newInstr;
```

```
            }
            // access to localvariables (storing) needs to be updated with a
                new index
            else if (Helper.IsOpcodeEqual(instr.OpCode, Helper.
                StoreLocalOpCodesArray))
            {
                int number = (instr.Operand as VariableDefinition).Index;

                // the instrution to use instead of the orginal "store"
                    instruction
                Instruction newInstr;
                switch (number)
                {
                    case 0:
                        newInstr = targetBody.CilWorker.Create(OpCodes.Stloc_0
                            );
                        break;
                    case 1:
                        newInstr = targetBody.CilWorker.Create(OpCodes.Stloc_1
                            );
                        break;
                    case 2:
                        newInstr = targetBody.CilWorker.Create(OpCodes.Stloc_2
                            );
                        break;
                    case 3:
                        newInstr = targetBody.CilWorker.Create(OpCodes.Stloc_3
                            );
                        break;
                    default:
                        if (number < 256)
                            newInstr = targetBody.CilWorker.Create(OpCodes.
                                Stloc_S, targetBody.Variables[number]);
                        else
                            newInstr = targetBody.CilWorker.Create(OpCodes.
                                Stloc, targetBody.Variables[number]);
                        break;
                }

                // access to localvariables needs to be updated with a new
                    index
                _instructionMapping.Add(instr, newInstr);
                targetBody.Instructions[index] = newInstr;
            }
        }

        CheckBranching(targetBody.Instructions);

    }

    /// <summary>
    /// Checks if the instructions in an InstructionCollection have references
         to
    /// instructions that has been replaced. If any such references are found,
         they are
    /// change to point at the instruction which has replaced the old
        instruction.
    /// </summary>
    /// <param name="instructions">The collection of instructions to check.</
        param>
    private void CheckBranching(InstructionCollection instructions)
    {
```

```
        foreach (Instruction instr in instructions)
        {
            if (Helper.IsOpcodeEqual(instr.OpCode, Helper.BranchOpCodesArray))
            {
                if (_instructionMapping.ContainsKey(instr.Operand as
                    Instruction)) //Instruction is in mapping - update
                    reference to the replacement instruction.
                {
                    Instruction mappedInstr = _instructionMapping[instr.
                        Operand as Instruction];
                    instr.Operand = mappedInstr;
                }
            }
            // special action needed for the switch instruction, as it has an
                array of instructions to branch to.
            else if (instr.OpCode.Equals(OpCodes.Switch))
            {
                Instruction[] switchInstructions = (instr.Operand as
                    Instruction[]);
                for (int i = 0; i < switchInstructions.Length; i++)
                {
                    if (_instructionMapping.ContainsKey(switchInstructions[i])
                        ) //Instruction is in mapping - update reference to
                        the replacement instruction.
                    {
                        Instruction mappedInstr = _instructionMapping[
                            switchInstructions[i]];
                        switchInstructions[i] = mappedInstr;
                    }
                }
            }

        }
    }

    /// <summary>
    /// Adds a collection of ExceptionHandlers to a method.
    /// </summary>
    /// <param name="exceptionHandlerCollection">The collection of
        ExceptionHandlers to add.</param>
    /// <param name="targetBody">The method which the ExceptionHandlers should
         be added to.</param>
    private void AddHandlers(ExceptionHandlerCollection
        exceptionHandlerCollection, MethodBody targetBody)
    {
        foreach (ExceptionHandler exceptionHandler in
            exceptionHandlerCollection)
        {
            targetBody.ExceptionHandlers.Add(exceptionHandler);

            if (exceptionHandler.Type == ExceptionHandlerType.Catch) //Handler
                 is of type catch, import the type to be catched.
            {
                if (!(Helper.IsAssemblyInRefs(targetBody.Method.DeclaringType.
                    Module.AssemblyReferences, exceptionHandler.CatchType) ||
                    Helper.IsAssemblyTarget((exceptionHandler.CatchType),
                    targetBody.Method.DeclaringType)))
                    YIIHAW.Output.OutputFormatter.AddWarning("It is not
                        possible to type check the use of '" +
                        exceptionHandler.CatchType.FullName + "'. Please make
                        sure that this class is available from the target
                        assembly.");
```

```
                if (Helper.IsAssemblyTarget(exceptionHandler.CatchType,
                    targetBody.Method.DeclaringType))
                    exceptionHandler.CatchType = Helper.FindLocalType(
                        targetBody.Method.DeclaringType, exceptionHandler.
                        CatchType);
                else
                    exceptionHandler.CatchType = targetBody.Method.
                        DeclaringType.Module.Import(exceptionHandler.CatchType
                        );
            }

            Instruction instr = exceptionHandler.FilterStart;
            if (instr != null && _instructionMapping.ContainsKey(instr))
                exceptionHandler.FilterStart = _instructionMapping[instr];

            instr = exceptionHandler.FilterEnd;
            if (instr != null && _instructionMapping.ContainsKey(instr))
                exceptionHandler.FilterEnd = _instructionMapping[instr];

            instr = exceptionHandler.HandlerStart;
            if (instr != null && _instructionMapping.ContainsKey(instr))
                exceptionHandler.HandlerStart = _instructionMapping[instr];

            instr = exceptionHandler.HandlerEnd;
            if (instr != null && _instructionMapping.ContainsKey(instr))
                exceptionHandler.HandlerEnd = _instructionMapping[instr];

            instr = exceptionHandler.TryStart;
            if (instr != null && _instructionMapping.ContainsKey(instr))
                exceptionHandler.TryStart = _instructionMapping[instr];

            instr = exceptionHandler.TryEnd;
            if (instr != null && _instructionMapping.ContainsKey(instr))
                exceptionHandler.TryEnd = _instructionMapping[instr];
        }
    }

    /// <summary>
    /// Copies the advice code into the new target, and takes action on
        special instructions in the advice.
    /// </summary>
    /// <param name="target">The target to insert into.</param>
    /// <param name="advice">The advice that should be inserted.</param>
    /// <param name="worker">The CilWorker of the old targetBody.</param>
    protected void HandleAdvice(MethodDefinition target, MethodDefinition
        advice, CilWorker worker)
    {
        UpdateLocalsInAdviceBody(advice.Body, target.Body);

        //pass through the advice instructions, either taking action on them,
        //adding them to the new target body, or both.
        for (int index = 0; index < advice.Body.Instructions.Count; index++)
        {
            Instruction curInstr = advice.Body.Instructions[index]; // current
                instruction
            if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
                MethodCallOpCodesArray))
                HandleMethodCall(curInstr, target, advice, worker);

            else if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
                LoadArgOpCodesArray))
```

```
            HandleLoadArgInstruction(curInstr, target, advice);

        else if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
            StoreArgOpCodesArray))
            HandleStoreArgInstruction(curInstr, target, advice);

        else if (curInstr.OpCode.Equals(OpCodes.Initobj))
            HandleInitObject(target.ReturnType.ReturnType, target.Body.
                CilWorker, advice.Body.CilWorker, curInstr);

        else if (curInstr.OpCode.Equals(OpCodes.Newobj))
            HandleNewobj(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Newarr))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Box))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
            UnBoxOpCodesArray))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Castclass))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
            LoadFieldOpCodesArray) || Helper.IsOpcodeEqual(curInstr.OpCode
            , Helper.StoreFieldOpCodesArray))
            HandleFieldReferences(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Isinst))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Constrained))
            HandleConstrained(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Ldobj))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Stobj))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Cpobj))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Mkrefany))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Refanytype))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Refanyval))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (curInstr.OpCode.Equals(OpCodes.Sizeof))
            HandleTypeImport(curInstr, target, advice, worker);

        else if (Helper.IsOpcodeEqual(curInstr.OpCode,Helper.
            ElementLoadAndStoreWithTokenArray))
            HandleTypeImport(curInstr, target, advice, worker);
```

```
            else if (curInstr.OpCode.Equals(OpCodes.Ldtoken))
            {
                if (curInstr.Operand is MethodReference)
                    HandleMethodReference(curInstr, target, advice, worker);
                else if (curInstr.Operand is TypeReference)
                    HandleTypeImport(curInstr, target, advice, worker);
                else
                    HandleFieldReferences(curInstr, target, advice, worker);
            }
            else
                target.Body.CilWorker.Append(curInstr);
        }
    }

    /// <summary>
    /// Handles instructions which has an operand of type MethodReference.
    /// The method reference is update, so that it works in the target
        assembly.
    /// </summary>
    /// <param name="curInstr">The instruction which has the MethodReference
        as operand.</param>
    /// <param name="target">The target method.</param>
    /// <param name="advice">The advice method.</param>
    /// <param name="worker">A CilWorker.</param>
    private void HandleMethodReference(Instruction curInstr, MethodDefinition
        target, MethodDefinition advice, CilWorker worker)
    {
        MethodReference methodRef = curInstr.Operand as MethodReference;

        // check that none of the generic parameters defined on the advice
            class or the advice method are passed as argument to other method
            - this is not allowed as the generic parameters do not exist at
            runtime (and would thus fail)
        foreach (GenericParameter genericParameter in methodRef.
            GenericParameters)
            if (advice.DeclaringType.GenericParameters.Contains(
                genericParameter) || advice.GenericParameters.Contains(
                genericParameter))
                throw new Exceptions.IllegalOperationException("The generics
                    parameters defined on the advice class or method can not
                    be used when calling methods ('" + advice.DeclaringType.
                    FullName + "." + advice.Name + "').");

        // check if the reference method has been added to the type the holds
            the method beind intercepted.
        if (_localMaps.MethodReferences.Lookup(target.DeclaringType,methodRef)
            != null)
        {
            methodRef = _localMaps.MethodReferences.Lookup(target.
                DeclaringType,methodRef);
            Instruction newInstr = worker.Create(curInstr.OpCode, methodRef);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }
        // Chech if the instruction is referencing a method defined in the
            aspect assembly. This is only allowed if the reference is not
            ambiguous and inserted into the target assembly.
        else if (methodRef.DeclaringType.Scope == advice.DeclaringType.Scope)
        {
            GlobalMapperEntry<MethodReference> mappedMethodRef = _globalMaps.
                MethodReferences.Lookup(methodRef);
            if (mappedMethodRef == null)
```

```
                throw new Exceptions.ConstructNotFoundException("Unable to
                    access the method '" + methodRef.Name + "' from '" +
                    target.Name + "', as '" + methodRef.Name + "' is not
                    defined in the target assembly. If this method should be
                    available in the target assembly, please specify that it
                    should be inserted into the target assembly using the
                    pointcut file.");
            else if (mappedMethodRef.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the method '" + methodRef.Name + "' from '" +
                    target.Name + "'. The reference is ambiguous, as the
                    method is inserted at multiple locations.");
            else
            {
                Instruction newInstr = worker.Create(curInstr.OpCode,
                    mappedMethodRef.Reference);
                _instructionMapping.Add(curInstr, newInstr);
                target.Body.CilWorker.Append(newInstr);
            }
        }
        // The refered method is not in the aspect assembly.
        // Check if it is in an assembly known by the target,
        // and else add an reference to the unknown assembly by importing the
            method reference.
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, methodRef.DeclaringType) || Helper.
                IsAssemblyTarget(methodRef.DeclaringType, target.DeclaringType
                )))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                    to type check the use of '" + methodRef.DeclaringType.
                    FullName + "'. Please make sure that this class is
                    available from the target assembly.");

            if (Helper.IsAssemblyTarget(methodRef.DeclaringType, target.
                DeclaringType))
                methodRef = Helper.FindLocalMethod(target.DeclaringType,
                    methodRef);
            else
                methodRef = target.DeclaringType.Module.Import(methodRef);

            Instruction newInstr = worker.Create(curInstr.OpCode, methodRef);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }
    }

    /// <summary>
    /// Handles the .constrained instruction.
    /// The instruction might be used on the generic returntype parameter in
        the advice,
    /// which will not be generic after the weaving. In that case the operand
        in the instruction
    /// should be changed to the new type of the parameter.
    /// </summary>
    /// <param name="curInstr">The .constrained instruction.</param>
    /// <param name="target">The target method.</param>
    /// <param name="advice">The advice method.</param>
    /// <param name="worker">A CilWorker.</param>
    private void HandleConstrained(Instruction curInstr, MethodDefinition
        target, MethodDefinition advice, CilWorker worker)
```

```
{
    TypeReference typeRef = curInstr.Operand as TypeReference;
    // if it is not a generic type, just handle it as any other
        instruction with a typeReference operand.
    if (!(typeRef is GenericParameter))
    {
        HandleTypeImport(curInstr, target, advice, worker);
        return;
    }
    // check that it is the generic return type from the advice.
    if (advice.ReturnType.ReturnType == typeRef)
    {
        curInstr.Operand = target.ReturnType.ReturnType;
        target.Body.CilWorker.Append(curInstr);
        return;
    }

        target.Body.CilWorker.Append(curInstr);

}

/// <summary>
/// Handles instructions which has an operand of type FieldReference.
/// The field reference is update, so that it works in the target assembly
    .
/// </summary>
/// <param name="curInstr">The instruction which has the FieldReference as
     operand.</param>
/// <param name="target">The target method.</param>
/// <param name="advice">The advice method.</param>
/// <param name="worker">A CilWorker.</param>
private void HandleFieldReferences(Instruction curInstr, MethodDefinition
    target, MethodDefinition advice, CilWorker worker)
{
    FieldReference fieldRef = curInstr.Operand as FieldReference;

    // check if the field that is being referred is located in the same
        assembly as the advice.
    if (fieldRef.DeclaringType == advice.DeclaringType) // the call is to
        a method defined in the same class as the aspect method - map the
        method call to the method that was created in the target type
        during the introduction phase
    {
        FieldReference mappedField = _localMaps.FieldReferences.Lookup(
            target.DeclaringType, fieldRef);
        if (mappedField != null) // the method was found in the mapper -
            insert a call to this method
        {
            Instruction newInstr = worker.Create(curInstr.OpCode,
                mappedField);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }
        else // field was not found in the mapper - throw an exception
            throw new Exceptions.ConstructNotFoundException("Unable to
                access field '" + fieldRef.Name + "' from '" + target.Name
                + "', as '" + fieldRef.Name + "' is not defined in the
                target assembly. If this field should be available in the
                target assembly, please specify that this field should be
                inserted into the target assembly using the pointcut
                specification.");
    }
```

```
        // check if the field referenced is in the aspect assembly.
        else if (fieldRef.DeclaringType.Scope == advice.DeclaringType.Scope)
            // this is a reference to a field inside the aspect assembly, but
            outside the declaring type of the aspect method currently being
            inserted - check if the field reference is ambiguous
        {
            GlobalMapperEntry<FieldReference> mappedField = _globalMaps.
                FieldReferences.Lookup(fieldRef);
            if (mappedField == null)
                throw new Exceptions.ConstructNotFoundException("Unable to
                    access field '" + fieldRef.Name + "' from '" + target.Name
                    + "', as '" + fieldRef.Name + "' is not defined in the
                    target assembly. If this field should be available in the
                    target assembly, please specify that this field should be
                    inserted into the target assembly using the pointcut
                    specification.");
            else if (mappedField.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    access field '" + fieldRef.Name + "' from '" + target.Name
                    + "'. The reference is ambiguous, as the field is
                    inserted at multiple locations.");
            else
            {
                Instruction newInstr = worker.Create(curInstr.OpCode,
                    mappedField.Reference);
                _instructionMapping.Add(curInstr, newInstr);
                target.Body.CilWorker.Append(newInstr);
            }
        }
        // The refered field is not in the aspect assembly.
        // Check if it is in an assembly known by the target,
        // and else add an reference to the unknown assembly by importing the
            field reference.
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, fieldRef.DeclaringType) || Helper.
                IsAssemblyTarget(fieldRef.DeclaringType, target.DeclaringType)
                ))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                    to type check the reference to '" + fieldRef.DeclaringType
                    .FullName + "." + fieldRef.Name + "'. Please make sure
                    that this field is available from the target assembly.");

            if (Helper.IsAssemblyTarget(fieldRef.DeclaringType, target.
                DeclaringType))
                fieldRef = Helper.FindLocalField(target, fieldRef);
            else
                fieldRef = target.DeclaringType.Module.Import(fieldRef);

            Instruction newInstr = worker.Create(curInstr.OpCode, fieldRef);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }
    }

    /// <summary>
    /// Given an Instruction that has a TypeReference as operand, this type is
         imported into the target.
    /// The instruction should come from an advice.
    /// </summary>
    /// <param name="curInstr">An Instruction which has a TypeReference as
```

```
    operand.</param>
/// <param name="target">The target to insert it into.</param>
/// <param name="advice">The advice from where the instruction originates
    .</param>
/// <param name="worker">A CilWorker.</param>
private void HandleTypeImport(Instruction curInstr, MethodDefinition
    target, MethodDefinition advice, CilWorker worker)
{
    TypeReference typeRef = curInstr.Operand as TypeReference;

    if (typeRef is GenericParameter && typeRef == advice.ReturnType.
        ReturnType)
    {
        Instruction newInstr = worker.Create(curInstr.OpCode, target.
            ReturnType.ReturnType);
        _instructionMapping.Add(curInstr, newInstr);
        target.Body.CilWorker.Append(newInstr);
        return;
    }

    // check that none of the generic parameters defined on the advice
        class or the advice method are passed as argument to other method
        - this is not allowed as the generic parameters do not exist at
        runtime (and would thus fail)
    foreach (GenericParameter genericParameter in typeRef.
        GenericParameters)
        if (advice.DeclaringType.GenericParameters.Contains(
            genericParameter) || advice.GenericParameters.Contains(
            genericParameter))
            throw new Exceptions.IllegalOperationException("The generics
                parameters defined on the advice class or method can not
                be used when instantiating objects ('" + advice.
                DeclaringType.FullName + "." + advice.Name + "').");


    // Trying to instantiate a type defined in the aspect assembly. This
        is only allowed if the reference is not ambiguous.
    if (typeRef.Scope == advice.DeclaringType.Scope)
    {
        GlobalMapperEntry<TypeReference> mappedType = _globalMaps.
            TypeReferences.Lookup(typeRef);
        if (mappedType == null)
            throw new Exceptions.ConstructNotFoundException("Unable to
                access the class '" + typeRef.Name + "' from '" + target.
                Name + "', as '" + typeRef.Name + "' is not defined in the
                 target assembly. If this class should be available in the
                 target assembly, please specify that it should be
                inserted into the target assembly using the pointcut file.
                ");
        else if (mappedType.IsAmbiguousReference)
            throw new Exceptions.IllegalOperationException("Unable to
                access the class '" + typeRef.Name + "' from '" + target.
                Name + "'. The reference is ambiguous, as the class is
                inserted at multiple locations.");
        else
        {
            Instruction newInstr = worker.Create(curInstr.OpCode,
                mappedType.Reference);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }
    }
```

```
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, typeRef) || Helper.IsAssemblyTarget(
                typeRef, target.DeclaringType)))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                    to type check the instantiation of '" + typeRef.FullName +
                     "'. Please make sure that this class is available from
                    the target assembly.");

            if (Helper.IsAssemblyTarget(typeRef, target.DeclaringType))
                typeRef = Helper.FindLocalType(target.DeclaringType, typeRef);
            else
                typeRef = target.DeclaringType.Module.Import(typeRef);

            Instruction newInstr = worker.Create(curInstr.OpCode, typeRef);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }
    }

    /// <summary>
    /// Takes a Newobj Instruction from the advice, and makes sure that it
        will work in the target.
    /// </summary>
    /// <param name="curInstr">The Newobj Instruction.</param>
    /// <param name="target">The target to insert it into.</param>
    /// <param name="advice">The advice from where the instruction originates
        .</param>
    /// <param name="worker">A worker from the original target body.</param>
    private void HandleNewobj(Instruction curInstr, MethodDefinition target,
        MethodDefinition advice, CilWorker worker)
    {
        MethodReference methodRef = curInstr.Operand as MethodReference;
        // check that none of the generic parameters defined on the advice
            class or the advice method are passed as argument to other method
            - this is not allowed as the generic parameters do not exist at
            runtime (and would thus fail)
        if (methodRef is GenericInstanceMethod)
            foreach (GenericParameter genericParameter in (methodRef as
                GenericInstanceMethod).GenericArguments)
                if (advice.DeclaringType.GenericParameters.Contains(
                    genericParameter) || advice.GenericParameters.Contains(
                    genericParameter))
                    throw new Exceptions.IllegalOperationException("The
                        generics parameters defined on the advice class or
                        method can not be used when instantiating objects ('"
                        + advice.DeclaringType.FullName + "." + advice.Name +
                        "').");


        // Trying to instantiate a type defined in the aspect assembly. This
            is only allowed if the reference is not ambiguous.
        if (methodRef.DeclaringType.Scope == advice.DeclaringType.Scope)
        {
            GlobalMapperEntry<MethodReference> mappedMethod = _globalMaps.
                MethodReferences.Lookup(methodRef);
            if (mappedMethod == null)
                throw new Exceptions.ConstructNotFoundException("Unable to
                    instantiate type '" + methodRef.DeclaringType.Name + "'
                    from '" + target.Name + "', as '" + methodRef.
                    DeclaringType.Name + "' is not defined in the target
```

```
                assembly. If this type should be available in the target
                assembly, please specify that it should be inserted into
                the target assembly using the pointcut specification.");
            else if (mappedMethod.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    instantiate type '" + methodRef.DeclaringType.Name + "'
                    from '" + target.Name + "'. The reference is ambiguous, as
                     the type is inserted at multiple locations.");
            else
            {
                Instruction newInstr = worker.Create(curInstr.OpCode,
                    mappedMethod.Reference);
                _instructionMapping.Add(curInstr, newInstr);
                target.Body.CilWorker.Append(newInstr);
            }
        }
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, methodRef.DeclaringType) || Helper.
                IsAssemblyTarget(methodRef.DeclaringType, target.DeclaringType
                )))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                    to type check the instantiation of '" + methodRef.
                    DeclaringType.FullName + "'. Please make sure that this
                    class is available from the target assembly.");

            if (Helper.IsAssemblyTarget(methodRef.DeclaringType, target.
                DeclaringType))
                methodRef = Helper.FindLocalMethod(target.DeclaringType,
                    methodRef);
            else
                methodRef = target.DeclaringType.Module.Import(methodRef);

            Instruction newInstr = worker.Create(curInstr.OpCode, methodRef);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }


    }




    /// <summary>
    /// Updates "Store Argument" instructions to match the placement of the
        argument in the targetbody.
    /// </summary>
    /// <param name="curInstr">The instruction that is storing an argument.</
        param>
    /// <param name="target">The target where the argument has been moved to
        .</param>
    /// <param name="advice">The advice containing the instruction.</param>
    private void HandleStoreArgInstruction(Instruction curInstr,
        MethodDefinition target, MethodDefinition advice)
    {
        int number = advice.Parameters.IndexOf(curInstr.Operand as
            ParameterDefinition);

        Instruction newInstr;
```

```
        if (number < 256)
            newInstr = target.Body.CilWorker.Create(OpCodes.Starg_S, target.
                Parameters[number]);
        else
            newInstr = target.Body.CilWorker.Create(OpCodes.Starg, target.
                Parameters[number]);

        _instructionMapping.Add(curInstr, newInstr); //registering replacement
             of instruction
        target.Body.CilWorker.Append(newInstr);
    }

    /// <summary>
    /// Updates "Load Argument" instructions to match the placement of the
        argument in the targetbody.
    /// </summary>
    /// <param name="curInstr">The instruction that is loading an argument.</
        param>
    /// <param name="target">The target where the argument has been moved to
        .</param>
    /// <param name="advice">The advice containing the instruction.</param>
    private void HandleLoadArgInstruction(Instruction curInstr,
        MethodDefinition target, MethodDefinition advice)
    {
        Instruction newInstr;
        if (curInstr.OpCode.Equals(OpCodes.Ldarga) || curInstr.OpCode.Equals(
            OpCodes.Ldarga_S))
        {
            int number = advice.Parameters.IndexOf(curInstr.Operand as
                ParameterDefinition);
            if (number < 256)
                newInstr = target.Body.CilWorker.Create(OpCodes.Ldarga_S,
                    target.Parameters[number]);
            else
                newInstr = target.Body.CilWorker.Create(OpCodes.Ldarga, target
                    .Parameters[number]);
        }

        if (curInstr.OpCode.Equals(OpCodes.Ldarg) || curInstr.OpCode.Equals(
            OpCodes.Ldarg_S))
        {
            int number = advice.Parameters.IndexOf(curInstr.Operand as
                ParameterDefinition);
            if (number < 256)
                newInstr = target.Body.CilWorker.Create(OpCodes.Ldarg_S,
                    target.Parameters[number]);
            else
                newInstr = target.Body.CilWorker.Create(OpCodes.Ldarg, target.
                    Parameters[number]);
        }
        else
        {
            int number = Helper.GetNumberValue(curInstr);
            switch (number)
            {
                case 0:
                    newInstr = target.Body.CilWorker.Create(OpCodes.Ldarg_0);
                    break;
                case 1:
                    newInstr = target.Body.CilWorker.Create(OpCodes.Ldarg_1);
                    break;
                case 2:
```

```
                    newInstr = target.Body.CilWorker.Create(OpCodes.Ldarg_2);
                    break;
                case 3:
                    newInstr = target.Body.CilWorker.Create(OpCodes.Ldarg_3);
                    break;
                case 4:
                    newInstr = target.Body.CilWorker.Create(OpCodes.Ldarg_S,
                        target.Parameters[number - 1]);
                    break;
                default:
                    throw new Exceptions.IllegalOperationException("Error
                        while handling load of argument operation");
            }
        }
        _instructionMapping.Add(curInstr, newInstr); //registering replacement
             of instruction
        target.Body.CilWorker.Append(newInstr);
    }

    /// <summary>
    /// Replaces all load local and store local instruction in an advice, with
         a ldloc or
    /// stloc instruction. Both of these uses a VariableDefinition as operand.
    /// </summary>
    /// <param name="advice">The advice to update.</param>
    /// <param name="target">The target of the interception.</param>
    protected void UpdateLocalsInAdviceBody(MethodBody advice, MethodBody
        target)
    {
        Instruction instr;
        for (int index = 0; index < advice.Instructions.Count; index++)
        {
            instr = advice.Instructions[index];

            if (Helper.IsOpcodeEqual(instr.OpCode, Helper.
                LoadLocalOpCodesArray) && !Helper.IsOpcodeEqual(instr.OpCode,
                Helper.LoadAdressOpCodesArray))
            {
                int number = Helper.GetNumberValue(instr);

                //the instrution to use instead of the orginal "load"
                    instruction
                Instruction newInstr;
                newInstr = advice.CilWorker.Create(OpCodes.Ldloc, target.
                    Variables[number]);
                _instructionMapping.Add(instr, newInstr);
                advice.Instructions[index] = newInstr;
                UpdateNextAndPreviousReferences(instr, newInstr);
            }

            if (Helper.IsOpcodeEqual(instr.OpCode, Helper.
                StoreLocalOpCodesArray))
            {
                int number = Helper.GetNumberValue(instr);

                //the instrution to use instead of the orginal "store"
                    instruction
                Instruction newInstr;
                newInstr = advice.CilWorker.Create(OpCodes.Stloc, target.
                    Variables[number]);
                _instructionMapping.Add(instr, newInstr);
                advice.Instructions[index] = newInstr;
```

```
                UpdateNextAndPreviousReferences(instr, newInstr);
            }
        }

        CheckBranching(advice.Instructions);

    }

    /// <summary>
    /// Updates the references in a linkedList of Instructions, when replacing
        an instruction with a new one.
    /// </summary>
    /// <param name="instr">The instruction being replaced.</param>
    /// <param name="newInstr">The replacement instruction.</param>
    private void UpdateNextAndPreviousReferences(Instruction instr,
        Instruction newInstr)
    {
        newInstr.Next = instr.Next;
        newInstr.Previous = instr.Previous;

        if (newInstr.Next != null)
            newInstr.Next.Previous = newInstr;

        if (newInstr.Previous != null)
            newInstr.Previous.Next = newInstr;
    }

    /// <summary>
    /// Handles the initobj instruction. Special action is only needed if it
        tries
    /// to init the generic parameter that the advice might has as returntype.
    /// In that case the generic parameter type should be change to the return
        type of the target method.
    /// </summary>
    /// <param name="typeReference">The return type of the target method.</
        param>
    /// <param name="targetWorker">The target body CilWorker.</param>
    /// <param name="adviceWorker">The advice body CilWorker.</param>
    /// <param name="instr">The Initobj Instruction.</param>
    protected void HandleInitObject(TypeReference typeReference, CilWorker
        targetWorker, CilWorker adviceWorker, Instruction instr)
    {
        VariableDefinition initobjStore = null;
        Instruction curInstr = instr;
        // the initobj instruction takes a pointer from the stack.
        // Find the instruction that load this pointer onto the stack,
        // and get the variable that the pointer points to.
        while ((curInstr = curInstr.Previous) != null)
        {
            if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
                LoadAdressOpCodesArray))
            {
                initobjStore = curInstr.Operand as VariableDefinition;
                break;
            }
        }
        if (initobjStore.VariableType == adviceWorker.GetBody().Method.
            ReturnType.ReturnType) // The type that is being initialized is of
             the same type as the returntype of the advice
            if (typeReference.FullName.Equals("System.Void")) // the
                returntype of the target method is void
            {
```

```
                targetWorker.Remove(curInstr); //removes the load address
                    instruction that has previously been copied to the
                    targetbody.

                //Get variable used for storing, and make a check for any
                    other loads and copies of the variable.
                int number = initobjStore.Index;
                List<int> varIndexes = FixVoidLoadAndStores(instr, number,
                    adviceWorker);
                int variablesRemoved = 0;
                foreach (int index in varIndexes)
                    targetWorker.GetBody().Variables.RemoveAt(index -
                        variablesRemoved++);
            }
            else
            {

                //The type of local variable from the advice body is changed
                    to that of the return type of the target.
                initobjStore.VariableType = typeReference;
                int varNum = targetWorker.GetBody().Variables.IndexOf(
                    initobjStore);
                List<Instruction> copyInstructions = FindCopyLocalVar(varNum,
                    instr);
                foreach (Instruction startCopyInstr in copyInstructions)
                {
                    int index = Helper.GetNumberValue(startCopyInstr.Next);
                    targetWorker.GetBody().Variables[index].VariableType =
                        typeReference;
                }
                Instruction newInstr = targetWorker.Create(OpCodes.Initobj,
                    typeReference);
                _instructionMapping.Add(instr, newInstr);
                targetWorker.Append(newInstr);
            }
        else
            targetWorker.Append(instr);
    }

    /// <summary>
    /// Finds the instructions that copies a local variable to another
        localvariable.
    /// </summary>
    /// <param name="varNum">The index of the localvariable.</param>
    /// <param name="instr">The instruction to start from.</param>
    /// <returns>A list of the instructions where the copy is started (load
        instructions).</returns>
    private List<Instruction> FindCopyLocalVar(int varNum, Instruction
        curInstr)
    {
        //The local variables which is used to store the return value is
            registered in a list.
        List<int> localsToCheck = new List<int>();
        localsToCheck.Add(varNum);

        List<Instruction> result = new List<Instruction>();

        while (curInstr != null)
        {
            if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
                LoadLocalOpCodesArray))
                if (localsToCheck.Contains(Helper.GetNumberValue(curInstr)))
```

```
                    // it is one of the registered locals that is being loaded
                    , check to see if it gets copied.
                    if (Helper.IsOpcodeEqual(curInstr.Next.OpCode, Helper.
                        StoreLocalOpCodesArray))
                    {   //add it to the watchlist.
                        localsToCheck.Add(Helper.GetNumberValue(curInstr.Next)
                            );
                        //add the instruction to the result list
                        result.Add(curInstr);
                    }
            curInstr = curInstr.Next;
        }
        return result;
    }

    /// <summary>
    /// Checks the remaining instructions in method, if they access a local
        variable, which will not be used because it would just store "void".
    /// If there is an access right before a "ret", it is removed,
    /// and if the localvariable is copied to another localvariable this is
        registered,
    /// and the copying is removed.
    /// </summary>
    /// <param name="instr">The instruction from where to start - this
        instruction is not included in the check.</param>
    /// <param name="varNum">The current known local variable index, where it
        is known that the "void" value is stored</param>
    /// <param name="worker">The CilWorker which controls the body of the
        instructions</param>
    /// <returns>A list of integers that indicates which localVariables has
        been used to hold the "Void" value.</returns>
    private List<int> FixVoidLoadAndStores(Instruction instr, int varNum,
        CilWorker worker)
    {
        //The local variables which is used to store the return value is
            registered in a list.
        List<int> localsToCheck = new List<int>();
        localsToCheck.Add(varNum);

        while ((instr = instr.Next) != null)
        {

            if (Helper.IsOpcodeEqual(instr.OpCode, Helper.
                LoadLocalOpCodesArray))
            {
                //we remove the load instruction if it is right before a
                    return
                if (instr.Next.OpCode.Equals(OpCodes.Ret))
                {
                    _instructionMapping.Add(instr, instr.Next);
                    worker.Remove(instr);
                }
                //else if it is one of the registered locals that is being
                    loaded, there are two options.
                else if (localsToCheck.Contains(Helper.GetNumberValue(instr)))
                {
                    if (Helper.IsOpcodeEqual(instr.Next.OpCode, Helper.
                        StoreLocalOpCodesArray))
                    { //the local is being copied to a new local, we add it to
                         our watchlist.
                        int newIndex = Helper.GetNumberValue(instr.Next);
```

```
                        if (!localsToCheck.Contains(newIndex))
                            localsToCheck.Add(newIndex);

                        Instruction newInstr = worker.Create(OpCodes.Nop);
                        _instructionMapping.Add(instr, newInstr);
                        _instructionMapping.Add(instr.Next, newInstr);

                        worker.InsertBefore(instr, newInstr);
                        worker.Remove(instr);
                        worker.Remove(instr.Next);
                        newInstr.Previous.Next = newInstr;
                        newInstr.Next = instr.Next.Next;

                        instr = instr.Next;
                    }
                    //if the next opcode is Initobj it means that default(Type
                        ) is invoked with the address of the variable - in
                        this case both the load-address and the initobj can be
                         deleted.
                    else if (instr.Next.OpCode.Equals(OpCodes.Initobj))
                    {
                        Instruction newInstr = worker.Create(OpCodes.Nop);
                        _instructionMapping.Add(instr, newInstr);
                        _instructionMapping.Add(instr.Next, newInstr);

                        worker.InsertBefore(instr, newInstr);
                        worker.Remove(instr);
                        worker.Remove(instr.Next);
                        newInstr.Previous.Next = newInstr;
                        newInstr.Next = instr.Next.Next;

                        instr = instr.Next;
                    }
                    else
                        //The local is being used, which it shouldn't.
                        throw new Exceptions.IllegalOperationException("The
                            program is trying to work on the return value from
                             a target method that returns void");
                }
            }
        }
        localsToCheck.Sort();
        return localsToCheck;
    }

    /// <summary>
    /// Checks if a methodcall from the advice body can be inserted in the
        target.
    /// Also checks if it is one of the special method calls to the YIIHAW.API
        , and handles
    /// it if it is one of thoose.
    /// </summary>
    /// <param name="curInstr">The call Instruction.</param>
    /// <param name="target">The target of the interception.</param>
    /// <param name="advice">The advice from where the Instruction is
        originating.</param>
    /// <param name="worker">The worker of the original target body.</param>
    private void HandleMethodCall(Instruction curInstr, MethodDefinition
        target, MethodDefinition advice, CilWorker worker)
    {
        MethodReference methodRef = curInstr.Operand as MethodReference;
        TypeReference typeRef = methodRef.DeclaringType;
```

```
// Check if it is one of the special YIIHAW.API calls.
if (typeRef.FullName.Equals("YIIHAW.API.JoinPointContext") &&
    methodRef.Name.Equals("Proceed")) // this is a call to the Proceed
     method - replace this call with the body of the target method
    HandleProceed(curInstr, target, worker, advice.Body.CilWorker);
else if (typeRef.FullName.Equals("YIIHAW.API.JoinPointContext") &&
    methodRef.Name.Equals("GetTarget")) // this is a reference to the
    Target property - replace this reference with a "this" pointer
    HandleTarget(curInstr, target, advice.Body);
else if (typeRef.FullName.EndsWith("YIIHAW.API.JoinPointContext"))
{
    Instruction newInstr = curInstr;
    if (methodRef.Name.Equals("get_DeclaringType")) // the user is
        call to the DeclaringType property - replace this with a
        string describing the declaring type of the target method
        newInstr = target.Body.CilWorker.Create(OpCodes.Ldstr, target.
            DeclaringType.FullName);
    else if (methodRef.Name.Equals("get_Name")) // the user is call to
         the Name property - replace this with a string describing the
         name of the target method
        newInstr =target.Body.CilWorker.Create(OpCodes.Ldstr, target.
            Name);
    else if (methodRef.Name.Equals("get_ReturnType")) // the user is
        call to the ReturnType property - replace this with a string
        describing the return type of the target method
        newInstr = target.Body.CilWorker.Create(OpCodes.Ldstr, target.
            ReturnType.ReturnType.FullName);
    else if (methodRef.Name.Equals("get_AccessSpecifier")) // the user
         is call to the AccessSpecifier property - replace this with a
         string describing the access specifier of the target method
    {
        string accessSpecifier = "";
        if ((target.Attributes & MethodAttributes.Assem) ==
            MethodAttributes.Assem)
            accessSpecifier = "internal";
        else if ((target.Attributes & MethodAttributes.Public) ==
            MethodAttributes.Public)
            accessSpecifier = "public";
        else if ((target.Attributes & MethodAttributes.Private) ==
            MethodAttributes.Private)
            accessSpecifier = "private";
        else if ((target.Attributes & MethodAttributes.Family) ==
            MethodAttributes.Family)
            accessSpecifier = "protected";
        else
            accessSpecifier = "unknown";

        newInstr = target.Body.CilWorker.Create(OpCodes.Ldstr,
            accessSpecifier);
    }
    else if (methodRef.Name.Equals("get_IsStatic")) // the user is
        call to the IsStatic property - replace this with a boolean
        determining if the target method is static of not
        newInstr = target.Body.CilWorker.Create((target.IsStatic ?
            OpCodes.Ldc_I4_1 : OpCodes.Ldc_I4_0));
    else if (methodRef.Name.Equals("get_Arguments")) // the user is
        call to the Arguments property - replace this with a string
        describing the arguments for the target method
    {
        string arguments = "";
        foreach (ParameterDefinition paramDef in target.Parameters) //
```

```
                    run through all arguments and build up a string
                    representing the arguments
                {
                    if (arguments.Length > 0)
                        arguments += ",";

                    arguments += paramDef.ParameterType.FullName;
                }

                newInstr = target.Body.CilWorker.Create(OpCodes.Ldstr,
                    arguments);
            }
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }
        else // this is a call to some other method (i.e. a method not part of
            the API for YIIHAW)
        {
            // check that none of the generic parameters defined on the advice
                class or the advice method are passed as argument to other
                method - this is not allowed as the generic parameters do not
                exist at runtime (and would thus fail)
            if (methodRef is GenericInstanceMethod)
                foreach (GenericParameter genericParameter in (methodRef as
                    GenericInstanceMethod).GenericArguments)
                    if (advice.DeclaringType.GenericParameters.Contains(
                        genericParameter) || advice.GenericParameters.Contains
                        (genericParameter))
                        throw new Exceptions.IllegalOperationException("The
                            generics parameters defined on the advice class or
                            method can only be used to invoke default(T) or
                            the Proceed<T>() method.");

            // check calls to methods from same assembly as the advice.
            if (methodRef.DeclaringType == advice.DeclaringType) // the call
                is to a method defined in the same class as the aspect method
                - map the method call to the method that was created in the
                target type during the introduction phase
            {
                MethodReference mappedMethod = _localMaps.MethodReferences.
                    Lookup(target.DeclaringType, methodRef);
                if (mappedMethod != null) // the method was found in the
                    mapper - insert a call to this method
                {
                    Instruction newInstr = worker.Create(curInstr.OpCode,
                        mappedMethod);
                    _instructionMapping.Add(curInstr, newInstr);
                    target.Body.CilWorker.Append(newInstr);
                }
                else // method was not found in the mapper - throw an
                    exception
                    throw new Exceptions.ConstructNotFoundException("Unable to
                        invoke method '" + methodRef.Name + "' from '" +
                        target.Name + "', as '" + methodRef.Name + "' is not
                        defined in the target assembly. If this method should
                        be available in the target assembly, please specify
                        that this method should be inserted into the target
                        assembly using the pointcut specification.");
            }
            else if (methodRef.DeclaringType.Scope == advice.DeclaringType.
                Scope) // this is a call to a method inside the aspect
                assembly, but outside the declaring type of the aspect method
```

```
            currently being inserted - check if the method reference is
            ambiguous
        {
            GlobalMapperEntry<MethodReference> mappedMethod = _globalMaps.
                MethodReferences.Lookup(methodRef);
            if (mappedMethod == null)
                throw new Exceptions.ConstructNotFoundException("Unable to
                     invoke method '" + methodRef.Name + "' from '" +
                    target.Name + "', as '" + methodRef.Name + "' is not
                    defined in the target assembly. If this method should
                    be available in the target assembly, please specify
                    that this method should be inserted into the target
                    assembly using the pointcut specification.");
            else if (mappedMethod.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    invoke method '" + methodRef.Name + "' from '" +
                    target.Name + "'. The reference is ambiguous, as the
                    method is inserted at multiple locations.");
            else
            {
                Instruction newInstr = worker.Create(curInstr.OpCode,
                    mappedMethod.Reference);
                _instructionMapping.Add(curInstr, newInstr);
                target.Body.CilWorker.Append(newInstr);
            }
        }
        else
        {
            if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                AssemblyReferences, methodRef.DeclaringType) || Helper.
                IsAssemblyTarget(methodRef.DeclaringType, target.
                DeclaringType)))
                YIIHAW.Output.OutputFormatter.AddWarning("It is not
                    possible to type check the call to '" + methodRef.
                    DeclaringType.FullName + "." + methodRef.Name + "'.
                    Please make sure that this method is available from
                    the target assembly.");

            if (Helper.IsAssemblyTarget(methodRef.DeclaringType, target.
                DeclaringType))
                methodRef = Helper.FindLocalMethod(target.DeclaringType,
                    methodRef);
            else
                methodRef = target.DeclaringType.Module.Import(methodRef);

            Instruction newInstr = worker.Create(curInstr.OpCode,
                methodRef);
            _instructionMapping.Add(curInstr, newInstr);
            target.Body.CilWorker.Append(newInstr);
        }

    }
}

/// <summary>
/// Handles a method call in the advice body which is a call to the
    Proceed method in the YIIHAW.API.
/// </summary>
/// <param name="curInstr">The call Instruction.</param>
/// <param name="target">The target of the weaving.</param>
/// <param name="worker">The CilWorker of the target.</param>
/// <param name="adviceWorker">The CilWorker of the advice.</param>
```

```
private void HandleProceed(Instruction curInstr, MethodDefinition target,
    CilWorker worker, CilWorker adviceWorker)
{
    if (_proceedInvoked)
        throw new Exceptions.IllegalOperationException("It is illegal to
            invoke Proceed() more than once within the same advice method
            ('" + adviceWorker.GetBody().Method.DeclaringType.FullName + "
            ." + adviceWorker.GetBody().Method.Name + "').");

    _proceedInvoked = true; // set flag indicating that the Proceed method
         has been invoked

    curInstr = curInstr.Next;

    //Inserting the old target body into the new one.
    foreach (Instruction targetInstr in worker.GetBody().Instructions)
        target.Body.CilWorker.Append(targetInstr);

    //If the return value of the call to "Proceed" is stored, special
        attention is needed
    if (Helper.IsOpcodeEqual(curInstr.OpCode, Helper.
        StoreLocalOpCodesArray))
    {
        int number = Helper.GetNumberValue(curInstr);

        //If the target method returns void, there is nothing to store.
        if (target.ReturnType.ReturnType.FullName.Equals("System.Void"))
        {
            //removing the storelocal instruction
            adviceWorker.Remove(curInstr);

            //As all stloc opcodes has been changed to "stloc operand",
                there will be an operand of type VariableDefinition.
            if (target.Body.Variables.Contains(curInstr.Operand as
                VariableDefinition)) //The variable has not yet been
                removed, and it should therefore be removed.
            {
                List<int> varIndexes = FixVoidLoadAndStores(curInstr,
                    number, adviceWorker);
                int variablesRemoved = 0;
                foreach (int index in varIndexes)
                    target.Body.Variables.RemoveAt(index -
                        variablesRemoved++);
            }
        }
        else
        //The return type is not void.
        {
            //The type of local variable from the advice body, is changed
                to that of the return type of the target
            VariableDefinition varDef = target.Body.Variables[number];
            varDef.VariableType = target.ReturnType.ReturnType;
            List<Instruction> copyInstructions = FindCopyLocalVar(number,
                curInstr);
            foreach (Instruction startCopyInstr in copyInstructions)
            {
                int index = Helper.GetNumberValue(startCopyInstr.Next);
                target.Body.Variables[index].VariableType = target.
                    ReturnType.ReturnType;
            }
        }
    }
```

```
        else if (curInstr.OpCode.Equals(OpCodes.Pop) && target.ReturnType.
            ReturnType.FullName.Equals("System.Void")) // The returnvalue from
             proceed is popped, but the target method returns void, so there
            is nothing to pop.
        {
            Instruction newInstr = adviceWorker.Create(OpCodes.Nop);
            adviceWorker.Replace(curInstr, newInstr);
            _instructionMapping.Add(curInstr, newInstr);
            UpdateNextAndPreviousReferences(curInstr, newInstr);
        }
    }

    /// <summary>
    /// Handles the call of the Target method in the YIIHAW.API.
    /// </summary>
    /// <param name="curInstr">The Instruction calling the method.</param>
    /// <param name="target">The target of the interception.</param>
    /// <param name="advice">The advice.</param>
    private void HandleTarget(Instruction curInstr, MethodDefinition target,
        MethodBody advice)
    {
        if (!target.IsStatic) //Only instance methods has a "this".
        {
            GenericInstanceMethod methodRef = curInstr.Operand as
                GenericInstanceMethod;
            if (methodRef.GenericArguments.Count > 0 && methodRef.
                GenericArguments[0].FullName.Equals(target.DeclaringType.
                FullName))
            {
                Instruction newInstr = target.Body.CilWorker.Create(OpCodes.
                    Ldarg_0);
                _instructionMapping.Add(curInstr, newInstr);
                target.Body.CilWorker.Append(newInstr);
            }
            else
                throw new Exceptions.IllegalOperationException("The type
                    defined for the GetTarget<T>() method does not match the
                    declaring type of the intercepted method.");
        }
        else
            throw new Exceptions.IllegalOperationException("You can not invoke
                 the Target property on a static method.");

    }

    /// <summary>
    /// Adds local variables to the target body and updates all references to
        these variables. Replaces return statements with branch operations.
    /// </summary>
    /// <param name="worker">A CIL worker for the original (unmodified) target
         body.</param>
    /// <param name="target">A reference to the target body.</param>
    /// <param name="advice">A reference to the advice body.</param>
    /// <param name="targetType">The declaring type of the body method.</param
        >
    protected void UpdateTargetBody(CilWorker worker, MethodBody target,
        MethodBody advice, TypeReference targetType)
    {
        // add the variables of the advice to the new target body (these are
            added first, as the user might not invoke the Proceed() method)
        foreach (VariableDefinition varDef in advice.Variables)
        {
```

```
        if (!(varDef.VariableType is GenericParameter) && !(varDef.
            VariableType is GenericInstanceType) && !(varDef.VariableType
            is ArrayType && (varDef.VariableType as ArrayType).ElementType
             is GenericInstanceType)) // Generic parameters do not need to
             be imported
        {
            if (varDef.VariableType is ArrayType)
                HandleArrayVar(varDef, advice.Method, target.Method);
            else if (varDef.VariableType.Scope == advice.Method.
                DeclaringType.Scope)
            {
                GlobalMapperEntry<TypeReference> mappedType = _globalMaps.
                    TypeReferences.Lookup(varDef.VariableType);
                if (mappedType == null)
                    throw new Exceptions.ConstructNotFoundException("
                        Unable to access the type '" + varDef.VariableType
                        .Name + "' from '" + target.Method.Name + "', as '
                        " + varDef.VariableType.Name + "' is not defined
                        in the target assembly. If this type should be
                        available in the target assembly, please specify
                        that it should be inserted into the target
                        assembly using the pointcut specification.");
                else if (mappedType.IsAmbiguousReference)
                    throw new Exceptions.IllegalOperationException("Unable
                         to access the type '" + varDef.VariableType.Name
                        + "' from '" + target.Method.Name + "'. The
                        reference is ambiguous, as the type is inserted at
                         multiple locations.");
                else
                    varDef.VariableType = mappedType.Reference;
            }
            else
            {
                if (!(Helper.IsAssemblyInRefs(target.Method.DeclaringType.
                    Module.AssemblyReferences, varDef.VariableType) ||
                    Helper.IsAssemblyTarget(varDef.VariableType, target.
                    Method.DeclaringType)))
                    YIIHAW.Output.OutputFormatter.AddWarning("It is not
                        possible to type check the use of '" + varDef.
                        VariableType.FullName + "'. Please make sure that
                        this class is available from the target assembly."
                        );

                if (Helper.IsAssemblyTarget(varDef.VariableType, target.
                    Method.DeclaringType))
                    varDef.VariableType = Helper.FindLocalType(target.
                        Method.DeclaringType, varDef.VariableType);
                else
                    varDef.VariableType = targetType.Module.Import(varDef.
                        VariableType);

            }
        }
        target.Variables.Add(varDef);
    }


    // add the variables of the original target to the new target body
    foreach (VariableDefinition varDef in worker.GetBody().Variables)
        target.Variables.Add(varDef);

    target.InitLocals = advice.InitLocals || worker.GetBody().InitLocals;
```

```
// indication of the number of variables that the advice has added to
    the
// new target body.
// The index of the old target variables is in the new target body now
     given as
// new_index = old_index + adviceLocalVariables.
int adviceLocalVariables = advice.Variables.Count;

// this last instruction is used, so that each return instruction in
    the target body
// can be changed to a "branch" to the last instruction in the target
    body.
Instruction lastInstr = worker.Create(OpCodes.Nop);
worker.Append(lastInstr);


// Pass through the target instructions.
for (int index = 0; index < worker.GetBody().Instructions.Count; index
    ++)
{
    Instruction instr = worker.GetBody().Instructions[index];

    //Access to localvariables needs to be updated with a new index
    if (Helper.IsOpcodeEqual(instr.OpCode, Helper.
        LoadLocalOpCodesArray) && !Helper.IsOpcodeEqual(instr.OpCode,
        Helper.LoadAdressOpCodesArray))
    {
        int number = Helper.GetNumberValue(instr) +
            adviceLocalVariables;

        //the instrution to use instead of the orginal "load"
            instruction
        Instruction newInstr;
        newInstr = worker.Create(OpCodes.Ldloc, target.Variables[
            number]);

        //The replacement of an instruction needs to be registered
        _instructionMapping.Add(instr, newInstr);
        worker.GetBody().Instructions[index] = newInstr;
        UpdateNextAndPreviousReferences(instr, newInstr);
    }
    //Access to localvariables needs to be updated with a new index
    else if (Helper.IsOpcodeEqual(instr.OpCode, Helper.
        StoreLocalOpCodesArray))
    {
        int number = Helper.GetNumberValue(instr) +
            adviceLocalVariables;

        //the instrution to use instead of the orginal "store"
            instruction
        Instruction newInstr;
        newInstr = worker.Create(OpCodes.Stloc, target.Variables[
            number]);

        //Access to localvariables needs to be updated with a new
            index
        _instructionMapping.Add(instr, newInstr);
        worker.GetBody().Instructions[index] = newInstr;
        UpdateNextAndPreviousReferences(instr, newInstr);
    }
```

```
            //The target method should no longer have return instructions, as
                the return
            // is done from the advice method.
            //each return is replaced by a "branch" to the last instruction in
                 the target body
            else if (instr.OpCode.Equals(OpCodes.Ret))
            {

                if (instr.Next == lastInstr)
                {
                    worker.Remove(instr);
                    _instructionMapping.Add(instr,lastInstr);
                    UpdateNextAndPreviousReferences(instr,lastInstr);
                }
                else
                {
                    Instruction newInstr = worker.Create(OpCodes.Br, lastInstr
                        );
                    _instructionMapping.Add(instr, newInstr);
                    worker.GetBody().Instructions[index] = newInstr;
                    UpdateNextAndPreviousReferences(instr, newInstr);
                }

            }
        }

        CheckBranching(worker.GetBody().Instructions);

    }

    /// <summary>
    /// Handles the case where a variable that should be inserted is of an
        arraytype.
    /// </summary>
    /// <param name="varDef">The variable that has the arraytype.</param>
    /// <param name="advice">The advice method.</param>
    /// <param name="target">The target method.</param>
    private void HandleArrayVar(VariableDefinition varDef, MethodDefinition
        advice, MethodDefinition target)
    {
        TypeReference typeRef = (varDef.VariableType as ArrayType).ElementType
            ;

        if (typeRef.Scope == advice.DeclaringType.Scope) // Trying to
            instantiate a type defined in the aspect assembly. This is only
            allowed if the reference is not ambiguous.
        {
            GlobalMapperEntry<TypeReference> mappedTypeEntry = _globalMaps.
                TypeReferences.Lookup(typeRef);

            if (mappedTypeEntry == null)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the type '" + typeRef.FullName + "' from '" +
                    target.DeclaringType.FullName + "." + target.Name + "', as
                     '" + typeRef.FullName + "' is not defined in the target
                    assembly. Please specify that this type should be
                    introduced using the pointcut file.");
            else if (mappedTypeEntry.IsAmbiguousReference)
                throw new Exceptions.IllegalOperationException("Unable to
                    access the type '" + typeRef.FullName + "' from '" +
                    target.DeclaringType.FullName + "." + target.Name + "'.
                    The reference is ambiguous, as the type is inserted at
```

```
                    multiple locations.");
                else
                    (varDef.VariableType as ArrayType).ElementType =
                        mappedTypeEntry.Reference;
            }
            else
            {
                if (!(Helper.IsAssemblyInRefs(target.DeclaringType.Module.
                    AssemblyReferences, typeRef) || Helper.IsAssemblyTarget(
                    typeRef, target.DeclaringType)))
                    YIIHAW.Output.OutputFormatter.AddWarning("It is not possible
                        to type check the instantiation of '" + typeRef.FullName +
                         "'. Please make sure that this class is available from
                        the target assembly.");

                if (Helper.IsAssemblyTarget(typeRef, target.DeclaringType))
                    (varDef.VariableType as ArrayType).ElementType = Helper.
                        FindLocalType(target.DeclaringType, typeRef);
                else
                    (varDef.VariableType as ArrayType).ElementType = target.
                        DeclaringType.Module.Import(typeRef);
            }
        }
    }
}
```

Modification.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Mono.Cecil;
using Mono.Cecil.Cil;

namespace YIIHAW.Weaver
{
    /// <summary>
    /// Used for the weaving part of modifications.
    /// </summary>
    public class Modification
    {
        /// <summary>
        /// Updates a target method, where the methods declaring type has changed
            basetype.
        /// If there is any calls to instance methods of the old basetype, they
            are changed
        /// to call the new basetype instead.
        /// Also type references to the old basetype are updated, and references
            to fields
        /// in the old basetype.
        /// </summary>
        /// <param name="methodDef">The method to update.</param>
        /// <param name="oldBaseType">The old basetype</param>
        /// <param name="newBaseType">The new basetype</param>
        public void ModifyMethod(MethodDefinition methodDef, TypeReference
            oldBasetype, TypeReference newBasetype)
        {
            if (methodDef.IsAbstract)
                return;

            foreach (Instruction curInstr in methodDef.Body.Instructions)
```

```
            {
                // update method references
                if (curInstr.Operand is MethodReference)
                {
                    MethodReference methodRef = curInstr.Operand as
                        MethodReference;
                    if (methodRef.DeclaringType == oldBasetype && !methodDef.
                        IsStatic)
                    {
                        TypeDefinition newType = Helper.FindLocalType(methodDef.
                            DeclaringType, newBasetype);
                        foreach (MethodDefinition newMethodDef in newType.
                            Constructors)
                            if (methodRef.Name.Equals(newMethodDef.Name))
                            {
                                curInstr.Operand = newMethodDef;
                                return;
                            }
                        foreach (MethodDefinition newMethodDef in newType.Methods)
                            if (methodRef.Name.Equals(newMethodDef.Name))
                            {
                                curInstr.Operand = newMethodDef;
                                return;
                            }
                    }
                }
                // update type references
                else if (curInstr.Operand is TypeReference)
                {
                    TypeReference typeRef = curInstr.Operand as TypeReference;
                    if (typeRef == oldBasetype)
                    {
                        curInstr.Operand = Helper.FindLocalType(methodDef.
                            DeclaringType, typeRef);
                        return;
                    }
                }
                // update field references
                else if (curInstr.Operand is FieldReference)
                {
                    FieldReference fieldRef = curInstr.Operand as FieldReference;
                    if (fieldRef.DeclaringType == oldBasetype)
                    {
                        TypeDefinition newType = Helper.FindLocalType(methodDef.
                            DeclaringType, newBasetype);
                        foreach (FieldDefinition newFieldDef in newType.Fields)
                            if (fieldRef.Name.Equals(newFieldDef.Name))
                            {
                                curInstr.Operand = newFieldDef;
                                return;
                            }
                    }
                }
            }
        }
    }
}
```

Mapper.cs

```
using System;
```

```
using System.Collections.Generic;
using System.Text;
using Mono.Cecil;
using System.Collections;

namespace YIIHAW.Weaver
{
    /// <summary>
    /// A special mapping (dictionary) that uses a key based on two object.
    /// The idea is that given a target type, and and aspect construct, it should
        be
    /// possible to find the copy of the aspect in the target.
    /// </summary>
    /// <typeparam name="T">The type of the value in the mapping.</typeparam>
    public class LocalMapper<T>
    {
        private Dictionary<int, T> map = new Dictionary<int, T>();

        /// <summary>
        /// Finds the value in the mapping that matches the key of the given
            arguments.
        /// </summary>
        /// <param name="typeRef">The type (target) where the value is placed.</
            param>
        /// <param name="aspect">The aspect that the value should be a copy of.</
            param>
        /// <returns>The value that the mapping returns based on the two arguments
            .</returns>
        public T Lookup(TypeReference typeRef, T aspect)
        {
            // calculate the key.
            int key = typeRef.GetHashCode() + aspect.GetHashCode();
            if (map.ContainsKey(key))
            {
                return map[key];
            }
            return default(T);
        }

        /// <summary>
        /// Checks if the mapping contains the given value.
        /// </summary>
        /// <param name="value">The value to look for.</param>
        /// <returns>A boolean indicating if the given value is present in the
            mapping.</returns>
        public bool ContainsValue(T value)
        {
            return map.ContainsValue(value);
        }

        /// <summary>
        /// Add a new value to the mapping.
        /// </summary>
        /// <param name="typeRef">The type in which the value is inserted.</param>
        /// <param name="aspect">The aspect that the value is a copy of.</param>
        /// <param name="target">The value to store.</param>
        public void Add(TypeReference typeRef, T aspect, T target)
        {
            // calculate the key.
            int key = typeRef.GetHashCode() + aspect.GetHashCode();
            map.Add(key, target);
        }
```

```
}

/// <summary>
/// A collection of LocalMapper.
/// Used to store the different kind of constructs that are inserted,
/// and which might need to be looked up again.
/// </summary>
public class LocalMapperCollection
{
    private LocalMapper<MethodDefinition> _methods;
    private LocalMapper<MethodReference> _methodReferences;
    private LocalMapper<FieldDefinition> _fields;
    private LocalMapper<FieldReference> _fieldReferences;

    public LocalMapper<MethodDefinition> Methods
    {
        get { return _methods; }
    }

    public LocalMapper<MethodReference> MethodReferences
    {
        get { return _methodReferences; }
    }

    public LocalMapper<FieldDefinition> Fields
    {
        get { return _fields; }
    }

    public LocalMapper<FieldReference> FieldReferences
    {
        get { return _fieldReferences; }
    }

    public LocalMapperCollection()
    {
        _methods = new LocalMapper<MethodDefinition>();
        _methodReferences = new LocalMapper<MethodReference>();
        _fields = new LocalMapper<FieldDefinition>();
        _fieldReferences = new LocalMapper<FieldReference>();

    }
}

/// <summary>
/// A special mapping (dictionary) that maps from T to a List of
    GlobalMapperEntry<T>.
/// </summary>
/// <typeparam name="T">The type used as key and as type parameter for
    GlobalMapperEntry.</typeparam>
public class GlobalMapper<T>
{
    private Dictionary<T, LinkedList<GlobalMapperEntry<T>>> map = new
        Dictionary<T, LinkedList<GlobalMapperEntry<T>>>();

    /// <summary>
    /// Add a mapping from a key to a list of GlobalMapperEntry where one of
        the
    /// GlobalMapperEntry will hold the value specified as parameter.
    /// </summary>
    /// <param name="key">The key to map from.</param>
    /// <param name="value">The value to insert into the GlobalMapperEntry
```

```
            which will be in the value list.</param>
        public void Add(T key, T value)
        {
            GlobalMapperEntry<T> newEntry = new GlobalMapperEntry<T>(value);


            if (!map.ContainsKey(key)) //the key has not been used before, create
                a new list.
                map.Add(key, new LinkedList<GlobalMapperEntry<T>>());
            else // there is already an entry in the list for this key.
                 // this means that the key is ambiguous.
            {
                newEntry.IsAmbiguousReference = true;
                foreach (GlobalMapperEntry<T> entry in map[key])
                    // as it is the GlobalMapperEntry that holds the ambiguous
                        information
                    // they all need to be updated.
                    entry.IsAmbiguousReference = true;
            }
            map[key].AddLast(newEntry);
        }

        /// <summary>
        /// Gets the first GlobalMapperEntry in the value list of a given key.
        /// </summary>
        /// <param name="key">The key to use in the lookup.</param>
        /// <returns>The first GlobalMapperEntry in the value list.
        /// If key is not in the mapping null is returned.</returns>
        public GlobalMapperEntry<T> Lookup(T key)
        {
            if (map.ContainsKey(key))
                return map[key].First.Value;

            return null;
        }

        /// <summary>
        /// Checks if a given value is in the mapping.
        /// </summary>
        /// <param name="value">The value to look for.</param>
        /// <returns>A boolean indicating if the value was found.</returns>
        public bool ContainsValue(T value)
        {
            foreach (KeyValuePair<T, LinkedList<GlobalMapperEntry<T>>> pair in map)
                foreach (GlobalMapperEntry<T> entry in pair.Value)
                    if (entry.Reference.Equals(value))
                        return true;

            return false;
        }
    }

    /// <summary>
    /// The type used as value used in the GlobalMapper.
    /// Holds a reference to an object of type T, and a boolean indicating
    /// if this entry is part of an ambiguous mapping.
    /// </summary>
    /// <typeparam name="T">The type T of the reference.</typeparam>
    public class GlobalMapperEntry<T>
    {
        private T _reference;
        private bool _ambiguousReference;
```

```
        public GlobalMapperEntry(T reference)
        {
            _reference = reference;
            _ambiguousReference = false;
        }

        public T Reference
        {
            get
            {
                return _reference;
            }
        }

        public bool IsAmbiguousReference
        {
            get
            {
                return _ambiguousReference;
            }
            set
            {
                _ambiguousReference = value;
            }
        }
    }

    /// <summary>
    /// A collection of GlobalMapper.
    /// Used to store the different kind of constructs that are inserted,
    /// and which might need to be looked up again.
    /// </summary>
    public class GlobalMapperCollection
    {
        private GlobalMapper<MethodDefinition> _methods;
        private GlobalMapper<MethodReference> _methodReferences;
        private GlobalMapper<FieldDefinition> _fields;
        private GlobalMapper<FieldReference> _fieldReferences;
        private GlobalMapper<TypeDefinition> _types;
        private GlobalMapper<TypeReference> _typeReferences;

        public GlobalMapper<MethodDefinition> Methods
        {
            get { return _methods; }
        }

        public GlobalMapper<MethodReference> MethodReferences
        {
            get { return _methodReferences; }
        }

        public GlobalMapper<FieldDefinition> Fields
        {
            get { return _fields; }
        }

        public GlobalMapper<FieldReference> FieldReferences
        {
            get { return _fieldReferences; }
        }
```

```
        public GlobalMapper<TypeDefinition> Types
        {
            get { return _types; }
        }

        public GlobalMapper<TypeReference> TypeReferences
        {
            get { return _typeReferences; }
        }

        public GlobalMapperCollection()
        {
            _methods = new GlobalMapper<MethodDefinition>();
            _methodReferences = new GlobalMapper<MethodReference>();
            _fields = new GlobalMapper<FieldDefinition>();
            _fieldReferences = new GlobalMapper<FieldReference>();
            _types = new GlobalMapper<TypeDefinition>();
            _typeReferences = new GlobalMapper<TypeReference>();

        }

    }

}
```

RecursiveDictionary.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace YIIHAW.Weaver
{
    /// <summary>
    /// A dictionary with recursive lookup.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public class RecursiveDictionary<T> : Dictionary<T,T>
    {
        public new T this[T t]
        {
            get
            {
                t = base[t];

                while (base.ContainsKey(t))
                    t = base[t];

                return t;
            }
            set
            {
                base[t] = value;
            }
        }
    }
}
```

Helper.cs

```
using System;
```

```
using System.Collections.Generic;
using System.Text;
using Mono.Cecil;
using Mono.Cecil.Cil;

namespace YIIHAW.Weaver
{
    public static class Helper
    {
        /// <summary>
        /// Checks to see if a given opcode is equal to one of the opcodes in an
            array of opcodes.
        /// </summary>
        /// <param name="opcode">The opcode to check</param>
        /// <param name="opcodes">The array of opcodes to check against</param>
        /// <returns>returns true if a match was found.</returns>
        internal static bool IsOpcodeEqual(OpCode opcode, params OpCode[] opcodes)
        {
            foreach (OpCode opc in opcodes)
            {
                if (opc.Equals(opcode))
                    return true;
            }
            return false;
        }

        internal readonly static OpCode[] LoadArgOpCodesArray = {
            OpCodes.Ldarg, OpCodes.Ldarg_0, OpCodes.Ldarg_1, OpCodes.Ldarg_2,
            OpCodes.Ldarg_3, OpCodes.Ldarg_S, OpCodes.Ldarga, OpCodes.Ldarga_S
        };



        internal readonly static OpCode[] StoreArgOpCodesArray = {
            OpCodes.Starg, OpCodes.Starg_S
        };

        internal readonly static OpCode[] LoadLocalOpCodesArray = {
            OpCodes.Ldloc, OpCodes.Ldloc_0, OpCodes.Ldloc_1, OpCodes.Ldloc_2,
            OpCodes.Ldloc_3, OpCodes.Ldloc_S, OpCodes.Ldloca, OpCodes.Ldloca_S
        };

        internal readonly static OpCode[] LoadAdressOpCodesArray = {
        OpCodes.Ldloca, OpCodes.Ldloca_S, OpCodes.Ldarga_S, OpCodes.Ldarga
        };


        internal readonly static OpCode[] StoreLocalOpCodesArray = {
            OpCodes.Stloc, OpCodes.Stloc_0, OpCodes.Stloc_1, OpCodes.Stloc_2,
            OpCodes.Stloc_3, OpCodes.Stloc_S
        };


        internal readonly static OpCode[] LoadConstantIntOpCodesArray = {
            OpCodes.Ldc_I4, OpCodes.Ldc_I4_0, OpCodes.Ldc_I4_1, OpCodes.Ldc_I4_2,
            OpCodes.Ldc_I4_3, OpCodes.Ldc_I4_4, OpCodes.Ldc_I4_5, OpCodes.Ldc_I4_6
                ,
            OpCodes.Ldc_I4_7, OpCodes.Ldc_I4_8, OpCodes.Ldc_I4_M1,OpCodes.Ldc_I4_S
                ,
            OpCodes.Ldc_I8
        };
```

```
internal readonly static OpCode[] LoadConstantFloatOpCodesArray = {
    OpCodes.Ldc_R4, OpCodes.Ldc_R8
};

internal readonly static OpCode[] UnBoxOpCodesArray = {
    OpCodes.Unbox, OpCodes.Unbox_Any
};

internal readonly static OpCode[] MethodCallOpCodesArray = {
    OpCodes.Call, OpCodes.Calli, OpCodes.Callvirt,
    OpCodes.Ldftn, OpCodes.Ldvirtftn, OpCodes.Jmp
};

internal readonly static OpCode[] BranchOpCodesArray = {
    OpCodes.Br, OpCodes.Br_S, OpCodes.Beq, OpCodes.Beq_S, OpCodes.Bne_Un,
    OpCodes.Bne_Un_S, OpCodes.Bge, OpCodes.Bge_S, OpCodes.Bge_Un,
    OpCodes.Bge_Un_S, OpCodes.Bgt, OpCodes.Bgt_S, OpCodes.Bgt_Un,
    OpCodes.Bgt_Un_S, OpCodes.Ble, OpCodes.Ble_S, OpCodes.Ble_Un,
    OpCodes.Ble_Un_S, OpCodes.Blt, OpCodes.Blt_S, OpCodes.Blt_Un,
    OpCodes.Blt_Un_S, OpCodes.Brfalse, OpCodes.Brfalse_S, OpCodes.Brtrue,
    OpCodes.Brtrue_S
};

internal readonly static OpCode[] ElementLoadAndStoreWithTokenArray = {
    OpCodes.Ldelema, OpCodes.Ldelem_Any, OpCodes.Stelem_Any
};

internal readonly static OpCode[] LoadFieldOpCodesArray = {
    OpCodes.Ldfld, OpCodes.Ldsfld, OpCodes.Ldsflda, OpCodes.Ldflda
};

internal readonly static OpCode[] StoreFieldOpCodesArray = {
    OpCodes.Stsfld, OpCodes.Stfld
};

/// <summary>
/// Finds the numeric value in a instruction (either in the opcode or the
    oprand).
/// </summary>
/// <param name="instr">The instruction to find the value in.</param>
/// <returns>The value found in the instruction.</returns>
internal static int GetNumberValue(Instruction instr)
{
    // check if the number is in the opcode.
    if (instr.OpCode.Equals(OpCodes.Ldloc_0) || instr.OpCode.Equals(
        OpCodes.Stloc_0) || instr.OpCode.Equals(OpCodes.Ldarg_0))
    {
        return 0;
    }
    else if (instr.OpCode.Equals(OpCodes.Ldloc_1) || instr.OpCode.Equals(
        OpCodes.Stloc_1) || instr.OpCode.Equals(OpCodes.Ldarg_1))
    {
        return 1;
    }
    else if (instr.OpCode.Equals(OpCodes.Ldloc_2) || instr.OpCode.Equals(
        OpCodes.Stloc_2) || instr.OpCode.Equals(OpCodes.Ldarg_2))
    {
        return 2;
    }
    else if (instr.OpCode.Equals(OpCodes.Ldloc_3) || instr.OpCode.Equals(
        OpCodes.Stloc_3) || instr.OpCode.Equals(OpCodes.Ldarg_3))
```

```
{
    return 3;
}
// check if the instruction is refering to a local variable, and get
    the index of the variable.
else if (instr.OpCode.Equals(OpCodes.Ldloc_S) || instr.OpCode.Equals(
    OpCodes.Stloc_S) ||
         instr.OpCode.Equals(OpCodes.Ldloc) || instr.OpCode.Equals(
             OpCodes.Stloc) ||
     instr.OpCode.Equals(OpCodes.Ldloca_S) || instr.OpCode.Equals(
         OpCodes.Ldloca))
{
    VariableDefinition vardef = instr.Operand as VariableDefinition;
    return vardef.Index;
}
// check if the opcode loads a constant.
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_0))
{
    return 0;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_1))
{
    return 1;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_2))
{
    return 2;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_3))
{
    return 3;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_4))
{
    return 4;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_5))
{
    return 5;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_6))
{
    return 6;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_7))
{
    return 7;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_8))
{
    return 8;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_M1))
{
    return -1;
}
else if (instr.OpCode.Equals(OpCodes.Ldc_I4_S) ||
         instr.OpCode.Equals(OpCodes.Ldc_I8) ||
         instr.OpCode.Equals(OpCodes.Ldc_I4))
{
    return (int)instr.Operand;
}
```

```
        throw new YIIHAW.Exceptions.InternalErrorException("The number in
            instruction " + instr.ToString() + " could not be found");
    }

    /// <summary>
    /// Checks if there is a reference to a specific assembly in a collection
        of AssemblyNameReferences.
    /// </summary>
    /// <param name="assemblyNameReferenceCollection">The collection to check
        up against.</param>
    /// <param name="the_assem">The assembly to look for.</param>
    /// <returns>Whether the AssemblyNameReference was found in the collection
         or not.</returns>
    public static bool IsAssemblyInRefs(AssemblyNameReferenceCollection
        assemblyNameReferenceCollection, TypeReference typeRef)
    {
        // find the AssemblyNameReference to compare against.
        AssemblyNameReference typeAssembly;
        if (typeRef.Scope is AssemblyNameReference)
            typeAssembly = typeRef.Scope as AssemblyNameReference;
        else if (typeRef.Module.Assembly.Name is AssemblyNameReference)
            typeAssembly = typeRef.Module.Assembly.Name as
                AssemblyNameReference;
        else
            throw new Exceptions.InternalErrorException("It was not possible
                to find an assemblyNameReference in type '" + typeRef.FullName
                 + "'.");

        // check if it can be found.
        foreach (AssemblyNameReference assem in
            assemblyNameReferenceCollection)
            if (assem.FullName.Equals(typeAssembly.FullName))
                return true;

        return false;
    }

    /// <summary>
    /// Checks if a type reference is to a type which is in the target
        assembly.
    /// </summary>
    /// <param name="typeRef">The type to check.</param>
    /// <param name="target">The target whose assembly to check up against.</
        param>
    /// <returns>A boolean indicating if the assembly of the type is the same
        as the target assembly.</returns>
    public static bool IsAssemblyTarget(TypeReference typeRef, TypeReference
        target)
    {
        // find the AssemblyNameReference to compare against.
        AssemblyNameReference assemblyName;
        if (typeRef.Scope is AssemblyNameReference)
            assemblyName = typeRef.Scope as AssemblyNameReference;
        else if (typeRef.Module.Assembly.Name is AssemblyNameReference)
            assemblyName = typeRef.Module.Assembly.Name as
                AssemblyNameReference;
        else
            throw new Exceptions.InternalErrorException("It was not possible
                to find an assemblyNameReference for type '" + typeRef.
                FullName + "'.");

        // do the comparing
```

```
        return assemblyName.FullName.Equals(target.Module.Assembly.Name.
            FullName);
    }

    /// <summary>
    /// Finds a field placed in the same type as a given method, which a given
         FieldReference is refering to.
    /// </summary>
    /// <param name="target">The target method placed in the type that should
        be searched.</param>
    /// <param name="fieldRef">The field reference for which the field should
        be found.</param>
    /// <returns>The field definition from the type.</returns>
    public static FieldDefinition FindLocalField(MethodDefinition target,
        FieldReference fieldRef)
    {
        if (target.DeclaringType.Module.Types.Contains(fieldRef.DeclaringType.
            FullName))
        {
            TypeDefinition targetType = target.DeclaringType.Module.Types[
                fieldRef.DeclaringType.FullName];
            foreach (FieldDefinition fieldDef in targetType.Fields)
                if (fieldRef.Name.Equals(fieldDef.Name))
                    return fieldDef;
        }
        throw new Exceptions.InternalErrorException("No matching type or field
             was found in FindLocalField");

    }

    /// <summary>
    /// Given an reference to a type and to a method, the type is found, and
        then the method is found in the type.
    /// </summary>
    /// <param name="target">The reference to the type.</param>
    /// <param name="methodRef">The method reference for which the method
        should be found.</param>
    /// <returns>The method definition from the type.</returns>
    public static MethodDefinition FindLocalMethod(TypeReference target,
        MethodReference methodRef)
    {
        // find the type.
        if (target.Module.Types.Contains(methodRef.DeclaringType.FullName))
        {
            TypeDefinition targetType = target.Module.Types[methodRef.
                DeclaringType.FullName];
            // find the method in the methods
            foreach (MethodDefinition methodDef in targetType.Methods)
                if (methodDef.Name.Equals(methodRef.Name))
                    return methodDef;
            // or find the method in the constructors.
            foreach (MethodDefinition methodDef in targetType.Constructors)
                if (methodDef.Name.Equals(methodRef.Name))
                    return methodDef;
        }
        throw new Exceptions.InternalErrorException("No matching type or field
             was found in FindLocalField");

    }


    /// <summary>
```

```
        /// Finds a type in the same assembly as a given type.
        /// </summary>
        /// <param name="target">The type placed in the assembly, where the other
            type should be found.</param>
        /// <param name="typeToFind">A reference to the type that should be found
            .</param>
        /// <returns>The type definition from the assembly.</returns>
        public static TypeDefinition FindLocalType(TypeReference target,
            TypeReference typeToFind)
        {
            if (target.Module.Types.Contains(typeToFind.FullName))
                return target.Module.Types[typeToFind.FullName];
            throw new Exceptions.InternalErrorException("No matching type was
                found in FindLocalType");
        }
    }
}
```