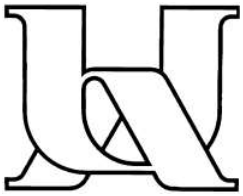

UNIVERSITEIT ANTWERPEN
Departement Wiskunde–Informatica

Dynamic Aspect Oriented Programming In .NET

Benny Van Aerschot

Academiejaar 2003 - 2004

Proefschrift ingediend tot het behalen van
de graad van Licentiaat in de Informatica

Promotor: Prof. Dr. Serge Demeyer
Co-Promotor: Prof. Dr. Bruce Watson
Begeleider: Andy Zaidman

Inleiding

Aspect-georiënteerd programmeren (AOP) is rond 1997 ontstaan [17]. AOP levert een mogelijkheid om aspecten van een applicatie, die niet gemodulariseerd kunnen worden, gebruik makend van de huidige software ontwikkelingsmethoden, toch onder te brengen in een afzonderlijke programmeerunit. Een dergelijke unit wordt een *aspect* genoemd.

Een *aspect* bevat alle informatie in verband met de tijdstippen waarop het in werking moet treden en welke code er moet uitgevoerd worden op dat moment. De software applicatie zelf hoeft dus geen informatie te hebben over het bestaan van *aspecten*.

Op een bepaald moment worden deze aspecten samengebracht met de rest van het programma. Dit proces wordt *weaving* genoemd. Afhankelijk van het moment waarop *weaving* gebeurt, spreekt men van *statische weaving* of *dynamische weaving*.

Statische weaving verwijst naar het samenvoegen van code tijdens het compilatieproces. Meestal wordt hiervoor gebruik gemaakt van een precompiler of parser. *Aspecten* worden dan vooraf geanalyseerd en verweven met de rest van programma. Het resultaat hiervan is opnieuw een sourcecode programma, dat doorgegeven kan worden aan een traditionele compiler.

Wanneer men op een later tijdstip aspecten wil toevoegen, verwijderen of aanpassen moet het compilatieproces opnieuw doorlopen worden. Ook moet de source code van het programma dat men wil aanpassen steeds ter beschikking zijn.

Het is veel makkelijker om deze aanpassingen door te voeren op het moment dat het programma in uitvoering is. Men spreekt dan van *dynamische weaving*.

Hoewel het meeste onderzoek zich tot nu toe gericht heeft op een statische AOP benadering, is er steeds meer en meer belangstelling voor een dynamische aanpak. Dit heeft geleid tot enkele implementaties voor de Java programmeertaal [25] [4].

In deze thesis wordt er op zoek gegaan naar een implementatie van dynamische AOP voor het Microsoft .NET Framework. Dit Framework heeft een architectuur die zeer gelijkaardig is aan de Java Virtuele Machine (JVM) en biedt bovendien extra mogelijkheden met betrekking tot het aanpassen en genereren van code op runtime.

De implementatie zou het mogelijk moeten maken om op runtime eenvoudig aspecten toe te voegen en te verwijderen. Bovendien wordt er gekeken of dit kan zonder interne aanpassingen te moeten doorvoeren aan het Framework zelf. Verder worden er zo min mogelijk eisen gesteld aan de programmeertaal waarin de *aspecten* gedefinieerd worden.

Abstract

Traditional Object-Oriented and procedural programming techniques often fall short when it comes to expressing behavior that does not fit naturally into a single program module. Aspect-Oriented Programming (AOP) is a new programming technique that addresses this problem. AOP focuses on behavior that cuts across the typical divisions of responsibility in a given programming model. These so-called cross-cutting concerns can be separated from the basic application's code and put into a single unit. Such a unit is termed an aspect.

This thesis presents a system that allows aspects to be added and removed to an existing application at runtime. The system works for any application that runs on the Microsoft .NET platform.

Acknowledgements

First of all I want to thank my promotor Prof. Dr. Serge Demeyer, who introduced me to software engineering and for giving me the opportunity to perform research in this area.

I want to thank his assistant, Hans Stenten, who led me through the first two months of my thesis.

His substitute, Andy Zaidman, guided me to the finish. His insightfulness and inspiring thoughts always put me right back on track.

It want to thank my parents and my brother Erwin, who supported and encouraged me every step of the way. I could not have done this without them.

Finally, I would like to say I am grateful for the helpful suggestions and comments I received from my fellow-students. A special thanks goes out to Bart Van Rompaey, Steven Hendrickx and Kelly Casal Mosteiro, for helping me when I needed it.

Contents

List of Figures

Chapter 1

Introduction

> *"If builders built buildings the way that programmers wrote programs, the first woodpecker that came along would destroy civilization"*
>
> *- Weinberg's Second Law*

Writing software applications has always been difficult. Still, through years of software development, there has been a significant evolution from monolithic, sequential, specialized programs, to programs consisting of clean, modular components, that are reusable in other systems. This in the end makes writing software easier [35] [9] [6].

With procedural programming techniques, programmers decomposed a computer problem into a series of steps or procedures. Each procedure would provide an abstraction for a specific subtask of the program. These procedures would then subsequently be executed and in doing so solve the bigger problem.

In 1967 the first Object-Oriented language called Simula [11] was developed. In the mid-1980's Object-Oriented Programming slowly entered mainstream applications, largely due to the influence of the C++ programming language [30]. This new programming paradigm offered an alternative way to model and design software.

Instead of modeling program flow, software is modeled using objects. Each object encapsulates a part of the problem. Objects can communicate with each other by exchanging messages. In this way the eventual program flow is implicitly defined.

Allowing to model the problem domain through the collaboration of objects enables better extensibility and reuse. This is a direct result of the modularization offered by Object-Orientation. Because each object represents a unit or module, with its own properties and behavior, changes or extensions to the system can often be kept local to only a few of these objects. Even more, objects used to solve a specific problem can later be reused to solve other problems.

Object-Oriented Programming quickly gained popularity. Unfortunately, it's not always easy, or possible, to put related behavior inside a separate unit. For instance, logging, security checks, transaction management... are concerns that will typically affect more than one module. These are called *cross-cutting* concerns, because they cut across the system's basic functionality. Such cross-cutting concerns result in *tangled* and *scattered* code that is hard to develop and maintain.

Aspect-Oriented Programming (AOP) is a new programming paradigm that enables clear separation of concerns. It introduces a new programming unit called an *aspect*, which is used to explicitly capture and modularize cross-cutting concerns. An illustration of how AOP works is shown below.

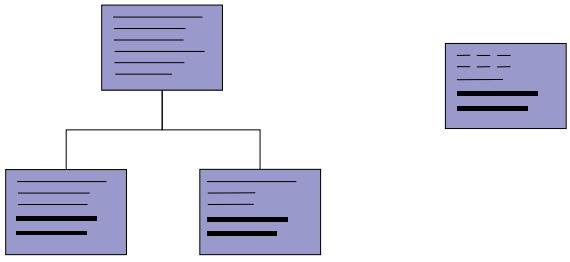

Figure 1.1: Without AOP

In Figure 1.1 a class diagram of a very simple Object-Oriented program is shown. Each line-type represents a different concern. The thick line represents a cross-cutting concern. With the use of Aspect-Oriented Programming techniques this concern can be separated from the rest of the program. The solution offered by AOP can be seen in Figure 1.2

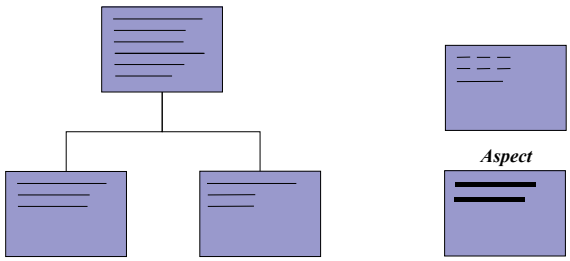

Figure 1.2: With AOP

Notice that the application is completely unaware of the existence of aspects. All the information that is needed resides in the aspect itself. It contains the code implementing the concern, together with a description of where the code needs to be inserted later. The process of merging aspect code with application code is termed *weaving*. Eventually, when the weaving process is complete, the code organization will again resemble Figure 1.1.

AOP complements Object-Oriented Programming and provides additional benefits due to the enhanced modularization:

- *Improved traceability and reusability*
- *Simplified incremental development.* Cross-cutting concerns are bundled in aspects and can be implemented independently. Information about when these aspects should be invoked is contained inside the description of the aspects itself, which implies that programmers developing the core system functionality do not need to be aware of the existence of aspect code.
- *Reduction in the number of source code lines*

Software architects can delay their design decisions if these decisions can potentially be implemented as aspects. Sometimes they can even be delayed till program runtime. This is where the dynamic facet comes into place. The runtime addition and removal of aspects is therefore called dynamic Aspect-Oriented Programming.

There are existing implementations of dynamic AOP for the Java language. The reason why Java is suitable to support dynamic AOP is that it runs on the Java Virtual Machine (JVM). Other languages, like C++, are compiled to an executable containing native code and do not provide a contained runtime environment. As a result runtime adaptation of a program written in these languages is a lot more difficult.

Recently the Microsoft .NET Framework has been introduced, which is conceptually very similarly to Java and its JVM. The .NET Framework contains a set of base class libraries and a runtime environment. Since the architecture of the Framework strongly resembles the JVM architecture it is a promising candidate for dynamic AOP.

Existing solutions for dynamic Aspect-Oriented Programming in Java heavily depend on a technique called *reflection* [27] [25]. This technique enables access to internal information of classes loaded into the JVM. Reflection is made possible through the use of metadata – data that describes other data. The particular type of metadata accessed by the Java reflection API[1], is the description of classes and objects within the JVM. This form of reflection is also called *introspection* [7] [16] [8].

The .NET Framework reflection not only enables retrieval of metadata (introspection), but even allows to emit metadata at runtime. This is a feature that the Java Reflection API does not offer. By using the .NET Reflection API's new types and even entire applications can be created at runtime.

Keeping this powerful reflection in mind, dynamic Aspect-Oriented Programming in the Microsoft .NET Framework shows great potential. This thesis provides an in-depth exploration of possible approaches and tries to

[1] Application Programming Interface

determine the most flexible solution for an AOP system, which enables run-time addition and removal of aspects.

The question that demands answering here is: "Is it possible to build a flexible, dynamic AOP system, while using only the tools and functionality provided by the Microsoft .NET framework? And on top of this, can this be done without the use of language extensions or internal changes to the .NET infrastructure?".

Chapter 2

Aspect-Oriented Programming

> *"To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. ..."*
>
> *- Dijkstra, A discipline of programming, 1976*

A software application is made out of many concerns, where one concern describes one particular area of interest. Concerns are the main reason for decomposing software into smaller parts or modules. But when developing a program using procedural or Object-Oriented Programming techniques, it can be decomposed into modules in only one way at a time. Concerns that are not aligned with this decomposition, remain scattered over several modules or tangled with other concerns within the same module. These concerns cut-across multiple modules, hence the name *cross-cutting* concern.

By adjusting the architecture of the system these cross-cutting concerns can be put inside modules, but in doing so other concerns become scattered and the same problem rises again. This problem is referred to as *the tyranny of the dominant decomposition* [31]. Aspect-Oriented Programming aims to solve this, by allowing cross-cutting concerns to be separated from the rest of the program and put inside a new programming unit, called an aspect. A typical cross-cutting concern is logging.

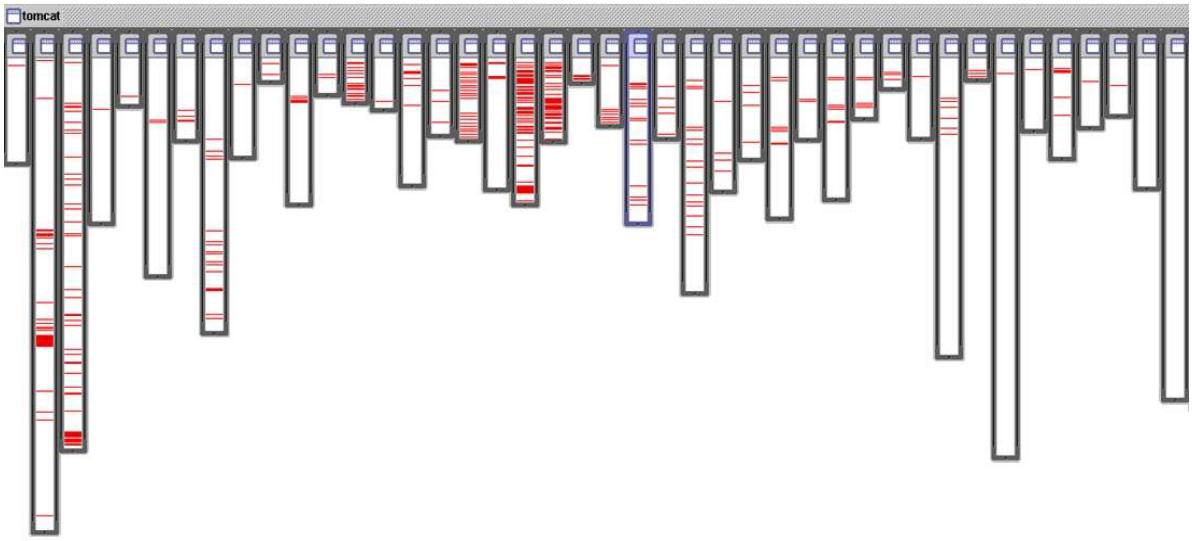

Figure 2.1: Logging in the Apache Tomcat Server [20]

In the illustration above each bar stands for a single module in the Apache Tomcat web server [33]. The code that is responsible for logging is represented by red lines. As can be seen here the logging code is scattered across almost the entire system, making it very hard to maintain. With the help of aspects this code can be put inside a single module.

2.1 Aspects

Cross-cutting concerns are not randomly distributed across an application. They have a clear purpose (what) and regular interaction points (where). This information is captured explicitly in the description of an aspect. To gain a better understanding of the contents of an aspect, some new terminology needs to be introduced.

A first definition is that of a *join-point*. A join-point is a well-defined point in the code at which the concern, addressed by the aspect, cross-cuts the application. This means that a join-point specifies when the aspect should be triggered. Typical join-points are method calls and constructor calls.

The distribution of join-points will follow a certain pattern, resulting from the fact that cross-cutting concerns are not randomly spread across a program. Therefore join-points are grouped in the definition of an aspect. A group of join-points is called a *pointcut*. The additional action to be taken when a join-point is reached, is termed *advice*.

With the help of these definitions an aspect can be defined as *a modular unit of cross-cutting behavior, containing:*

- *pointcuts (one ore many)*
- *advice (one or many) for each pointcut.*

2.2 Weaving

When all concerns are identified and bundled into aspects, there is still nothing that relates the original program to the obtained aspects. At some point these aspects need to be combined with the rest of the program. This process is called *weaving*, because the aspect code is actually woven in with the application code. Depending on the time at which the weaving takes place, different types of weaving can be distinguished.

2.2.1 Compile-Time Weaving

The easiest and therefore most popular weaving approach is compile-time weaving (also called *static weaving*). The source code of the original application is merged with the aspect code during compilation, resulting in a new program. This can be done using a preprocessor or an aspect weaver that is directly integrated with the compiler. The most widely known static aspect weaving tool is AspectJ [3].

2.2.1.1 AspectJ

AspectJ extends the Java language with new keywords for modeling aspects. It uses a pre-compiler that translates the aspect code into plain Java and

weaves it with the rest of the application code. After passing the precompiler, the merged code is handed to the normal Java compiler. An illustration of how static AOP works is presented in Figure 2.2.

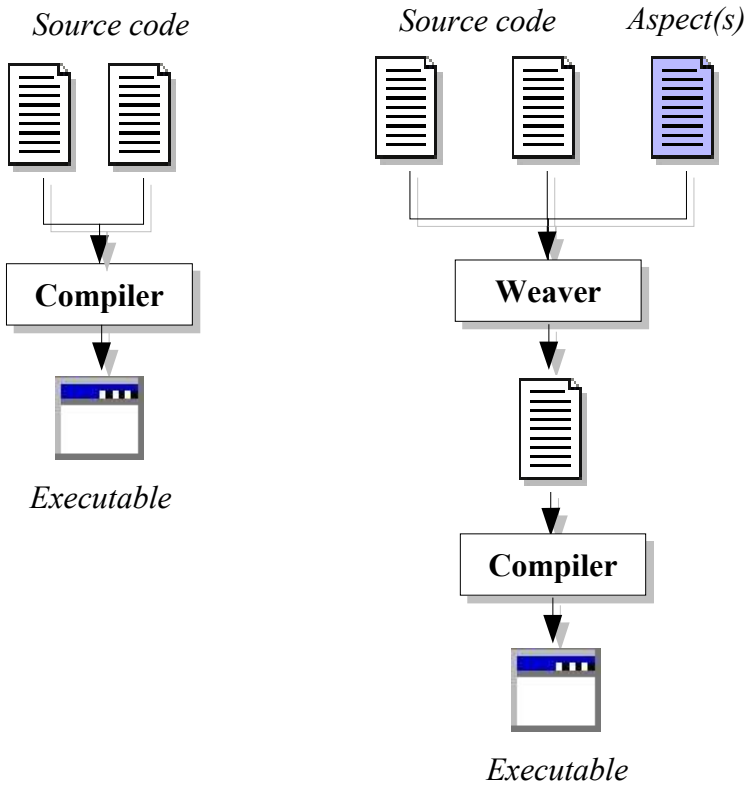

Figure 2.2: Static AOP

In Figure 2.3 a logging aspect in AspectJ is presented. Aspects in AspectJ are very similar to plain Java classes. They can contain methods and fields, just like normal classes. In addition, an aspect has to contain at least one advice and one pointcut.

The pointcut presented in Figure 2.3 represents any call to a public method in the `org.apache.tomcat` package. In AspectJ pointcuts can be named. Here it is associated with the name `publicInterface`. The description of the advice contains the keywords `after throwing`, which refines the

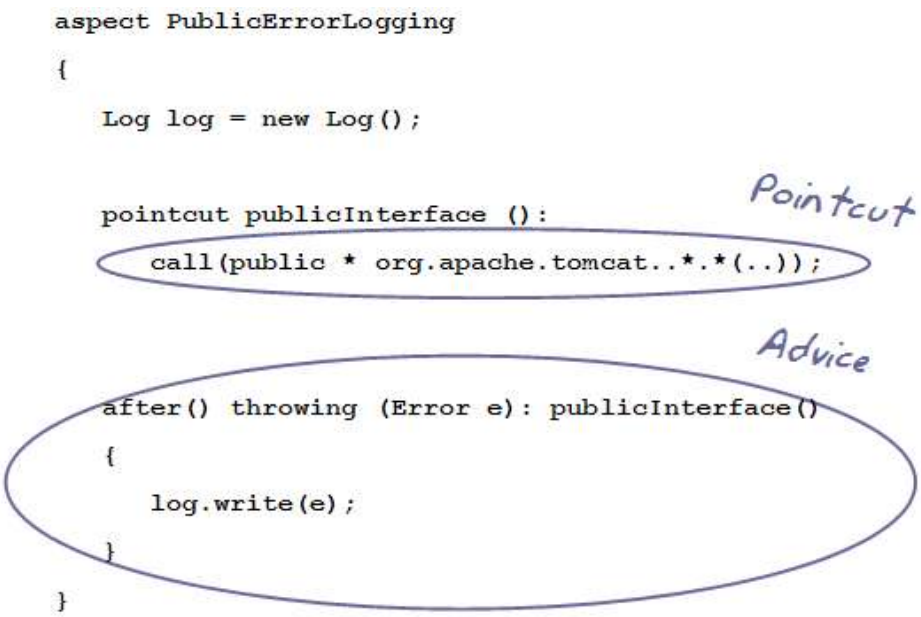

Figure 2.3: An Aspect in AspectJ

exact moment when the advice should be executed. In this case, the advice will be executed after an exception was thrown inside a method, matching the definition of the pointcut. When this happens, the captured error is written to a log file.

AspectJ was developed at Xerox Palo Alto Research Center. It was the first general purpose AOP language and weaver. After the announcement of AspectJ, numerous research projects have been set up to transfer the Aspect-Oriented Programming idea to other languages. This resulted in tools like:

- **AspectC++**, for the C++ programming language [12]
- **LOOM.NET**, a static aspect weaver for .NET [28]
- **AspectS**, for Squeak/Smalltalk [14]
- **AspectC#**, provides AspectJ-style aspect support to C# [18]
- **...**

A complete list of research projects, dynamic and static approaches, can be found on the AOSD website [1].

Static weaving tools have the benefit of producing well formed code, because all code needs to pass a compiler. As a result, only very few extra checks need to be performed at runtime. However, a major disadvantage is that the source code of the original program must be available. Moreover, adding and removing aspects also requires to recompile the entire application.

2.2.2 Load-Time Weaving

Weaving tools have been developed that allow to postpone the actual weaving of an application and aspects till load-time. Approaches like Binary Component Adaptation (BCA) and Java Object Instrumentation Environment (JOIE) have been developed for the Java language.

Both BCA and JOIE replace the original Java class loader with their own version. This allows to replace an entire class by another without the Java Virtual Machine knowing about it. With this approach the original source code is not required anymore. However, desired changes to a program have to be specified as changes to classes, which makes expressing cross-cutting concerns harder.

2.2.3 Runtime Weaving

In the evolution of software development there is, and has been, a growing trend to defer the binding of method calls to actual executable code for as long as possible [10]. In the early days programs were written in direct machine-level instructions. Here, method calls could be compared with jump instructions, which would branch to a specified label. Altering these labels or instructions, to influence the control flow of the program was difficult and error-prone.

Procedural programming relieved the programmer of the cumbersome task to specify exact labels, the program should jump to, by offering a higher level of abstraction.

Object-Oriented Programming (OOP) goes yet another step further. Polymorphism provides a common behavior and interface for related concepts and allows more specialized components to change a particular behavior. Control flow is thus implicitly defined and can be different, depending on the runtime type of certain objects.

However, the flexibility offered by OOP is still quite limited. Although the behavior of a program can vary, the possibilities in which it can differ is decided at compile-time. The ideal situation would be to postpone these decisions till runtime. Dynamic Aspect-Oriented Programming offers a solution here.

An ideal dynamic AOP solution should allow runtime addition and removal of aspects. Aspects can manipulate the existing program in such a way that the original program will exhibit entirely different behavior. This is particularly useful for applications for which future adjustments are obvious, but not yet implemented. If these modifications can be modeled as aspects, they can be developed and applied when they are actually needed.

Moreover, the changes made by aspects are transparent to the running system and do not require a shutdown. The importance and simplicity offered by this approach is illustrated by a real world example.

2.2.3.1 Real world example - an online store

An online store that runs 24-7 cannot afford to go offline for every new modification. Basically, every visit to a web store means a potential sale. The downtime – the time the site is offline and unreachable for customers – should thus be as short as possible, ideally zero. If the software that runs on the web server supports dynamic aspects, many of these adjustments can be made by using aspects.

For instance, the online shop owner can decide to give a 40% discount on all products, during the holiday season. This minor adjustment can easily be bundled in an aspect. Assuming for instance, that all available product objects have a `getPrice` method, a pointcut can be used to capture these methods. The corresponding advice can than incorporate the 40% reduction.

By using dynamic AOP the online store can keep receiving customers when this modification takes effect. At the end of the holiday period, the aspects that are no longer needed can be removed and the web store can continue its normal operation.

2.2.3.2 Other application domains

The previous example shows the use dynamic AOP for the implementation of business rules. Business rules implement company policies and are known to change frequently. The general form of a business rule is often: `if` *`condition`* `then` *`action`*. Dynamic AOP facilitates the modification and isolation of business rules in running applications.

A nice overview of application domains is presented in [13]. A first example is a load balancing aspect, as also presented in [21]. This aspect can replace a previous load balancing strategy with a better one depending on the current load of servers. Therefore in certain cases survival of aspects at run time is necessary to allow them to adapt to suitable policies according to execution time information.

Another use for dynamic AOP would be to gather information from long running applications. It can be a lot more useful and interesting to obtain this information without the need to restart the running application.

Web applications can benefit from dynamic AOP for the application of *hot fixes* [24]. A hot fix is an extension applied to a running application server to modify the behavior of a large number of running components.

Logging or method call tracing is another prominent use of dynamic AOP. This can be useful to obtain statistics about frequently used functionality, or to trace errors in a running application.

Another interesting example is the adaptation of mobile devices [24]. Mobile devices have very small memory and so the device is not able to have all the software components needed in various locations from the start. Therefore it should dynamically acquire the needed functionality it needs in a certain location and discard when location changes. This functionality has

often a cross-cutting character and so is realized as aspects.

2.2.3.3 PROSE

An example Java dynamic AOP system is PROSE (PROgrammable extenSions of sErvices) [25]. Its implementation is based on the Java Virtual Machine Debugger Interface (JVMDI). As a result, all applications that need to make use of PROSE have to be run in debug mode, which introduces a significant performance penalty.

Whenever a join-point is reached, the JVMDI passes control to PROSE, which then identifies the applicable aspects. Information about the currently executing program, known to the JVMDI is then passed to the appropriate advices. Once the execution of the advice methods has finished, control is again returned to the running application.

2.3 Related work

In what follows, existing solutions and research projects specifically targeted for the .NET platform are discussed in more detail.

AspectC# [18] offers static AOP solution. Unlike AspectJ, it does not introduce language extensions. Instead, it employs special .NET custom attributes[1] which are specified on top of methods, fields or classes. A separate XML file connects the aspects to a concrete context. AspectC# uses a preprocessor which uses the XML file to bind the aspects and application code at compile time.

Another static weaver is provided by LOOM.NET [28]. It provides some predefined aspects that can be connected with an application using a Graphical User Interface. Alternatively, aspects can be connected directly in the source code, again using .NET attributes. The user can define his own aspects using XML files.

[1] see Chapter 3 - 3.9 Attributes

The RAPIER-LOOM.NET [29] library makes it possible to create proxy classes at runtime. The weaver creates a new assembly at runtime which contains the original code completed with the aspect code. To reference code in the new assembly each constructor call is replaced by a call to the weaver. Arguments to the call are the original class followed by an arbitrary long enumeration of aspect classes.

In order to replace the constructor calls, the original code has to be available. Moreover all aspects need to be known at this time and the main method of the original application has to be rewritten. Another limitation to this approach is that all methods, that will be changed by one of the aspects, have to be declared virtual. In Java all methods are implicitly declared virtual, in C# and other .NET languages this property has to be stated explicitly.

Chapter 3

The Microsoft .NET Framework

This chapter will give more insight in the inner workings of the .NET Framework. It will introduce Intermediate Language code and explain how this code is JIT compiled to native code. The terms attributes, metadata and assembly will be defined and clarified. Also some attention is given to COM, the component model that preceded the .NET platform, because it is used in the implementation of the Aspect-Oriented system, discussed in the next chapter.

3.1 The Common Language Infrastructure

The standard supported languages in version 1.1 of the Framework are Visual Basic .NET, Visual C++, C# and J Script. To make it possible for other languages to use the .NET framework, a large part of the CLR has been standardized as the Common Language Infrastructure (CLI) [15]. The CLI architecture consists of 4 main parts:

- **The Common Type System (CTS)** provides a type system that supports the types and operations found in modern programming languages.
- **Metadata** is used by the CLI to describe and reference the types defined by the Common Type System. Metadata is thoroughly explained in Section 3.8.

- **The Common Language Specification (CLS)** is a set of rules that specifies when a language is compliant with the .NET platform. It specifies a subset of the CTS and a set of usage conventions. A CLS-compliant language is granted complete access to the .NET framework and interoperability with other compliant languages.

- **The Virtual Execution System (VES)** implements and enforces the CTS model. It is that part of the CLI where programs actually run in.

Figure 3.1 illustrates the Common Language Runtime. In the following sections the parts and inner-workings of the Microsoft .NET Framework are explained.

3.2 The Compilation And Execution Process

When writing a .NET application, the choice of programming language is left free to the programmer. The Microsoft .NET platform can offer this freedom through the use of a Common Intermediate Language (also called Microsoft Intermediate Language or MSIL, or just IL). Any language providing a compiler that complies to the Common Language Specification will produce this IL code.

The implication of using an Intermediate Language is that programming languages can freely interoperate. For example, a VB.NET class can inherit from a C# class, because at the Intermediate Language level all programming languages use the same instructions. The types used are those specified by the Common Type System.

Java also uses an intermediate language, called byte code. Java does not compile its source code (.JAVA) files directly into machine instructions. Instead, it uses an in-between platform-independent representation. The files containing these intermediate language instructions have a (.CLASS) extension.

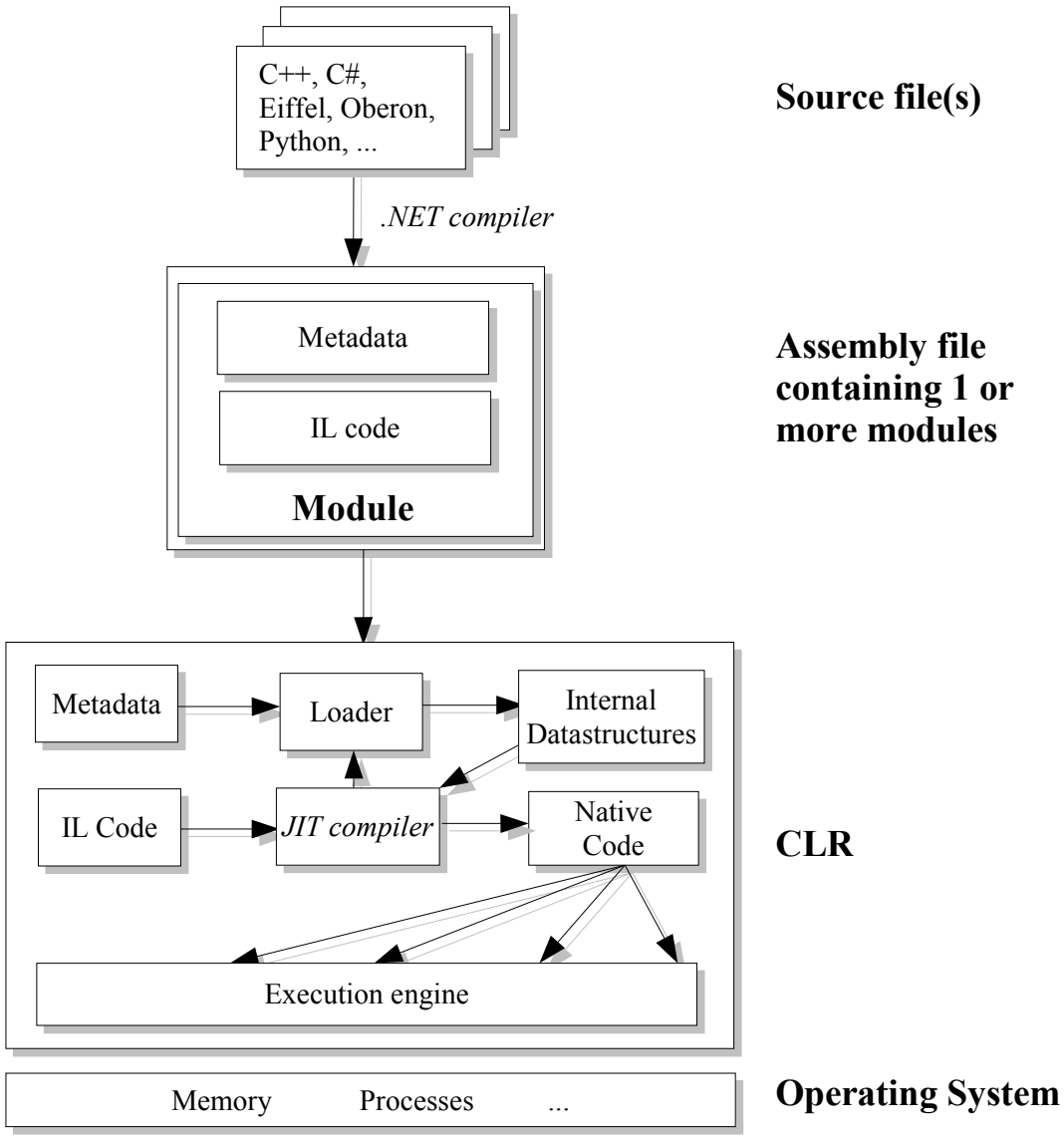

Figure 3.1: The Common Language Runtime

3.3 Assemblies

Passing the source files through a .NET compiler results in a .DLL or .EXE file. These files are called *assemblies* and form the basic building blocks of a .NET application. The name 'assembly' covers two different facets.

On one hand it denotes the physical form, namely the well known Portable Executable format (.DLL or .EXE file), as used by Microsoft Windows for a long time. On the other hand it refers to the logical content. An assembly logically contains the following elements:

- Manifest (mandatory, contains the assembly metadata)
- Type metadata, describing the types in the assembly (optional)
- Intermediate Language (IL) code, implementing the types (optional)
- Resources (optional, these are other resources, such as image files, ...)

3.4 JIT Compilation

MSIL code is not interpreted, which means that the CLR does not execute Intermediate Language code directly. Instead, the Common Language Runtime translates IL code to native machine code (like x86, IA64, ...) prior to execution. This process is called *Just-In-Time compilation* or *JIT compilation*.

Whenever the CLR loads a class type each method is replaced by stub code that transfers control to the JIT compiler. Upon actual execution of a method, the method's body is translated to native code. After the compilation has finished, the stub code is replaced with the address of the compiled code. Future executions of the method directly execute this cached version of the method body.

The loader is that part of the CLR that is responsible for loading assemblies, modules and types. When the JIT compiler encounters a type, it triggers the CLR loader, which will then load the type into memory. In doing so, the associated module and assembly are loaded too. By employing this

strategy, those parts of a program that are never used, never get loaded into memory either.

3.5 Managed & Unmanaged

Code that requires the Common Language Runtime to run, is called *managed code*. When the CLR executes this code, it is said to run in *managed execution mode* [5]. In this mode, the CLR is made responsible for JIT compilation, loading, garbage collection, etc. In other words, the Common Language Runtime has complete control over the running application.

At the other end is *unmanaged code*. In contrast with its managed counterpart, the CLR has no information about it. Unmanaged code is run in *unmanaged execution mode* and is rendered an opaque black box for the CLR. As a result, the CLR is incapable of handling garbage collection or any other service. All these responsibilities are thus left to the program itself. The only power left to the CLR is to start and stop its execution.

C# and VB.NET compilers emit managed code, while others, like the C++ compiler, can emit both unmanaged and managed code. The C++ compiler even allows to build mixed assemblies. If this is the case, the Common Language Runtime will switch between managed and unmanaged execution mode, when necessary.

3.6 COM

The previous software component technology endorsed by Microsoft was the Common Object Model (COM) [32] [5]. Although COM has been a successful component model, programming using COM was hard. Many of the work regarding object lifetime management and interoperability was left to the programmer. To make these things easier, Microsoft introduced .NET. However, there exists a limited backward compatibility with COM[1].

[1] a COM object may be used in .NET by implementing a *runtime callable wrapper* and .NET may use COM objects by calling a *COM callable wrapper*. Details of these possibilities are not discussed here

Microsoft also provides some unmanaged COM interfaces for inspection and manipulation of metadata and the CLR execution engine. The functionality offered by these interfaces proved to be very useful in the implementation of the Aspect Engine presented in the next chapter. These interfaces are divided into three groups:

- Unmanaged Metadata API[2]
- Debugger API
- Profiler API

Both the Unmanaged Metadata API and the Profiler API are used in the implementation of the profiler, which forms the core of the AOP system.

3.7 The Profiler

The Microsoft .NET Software Development Kit (SDK) v1.1 is shipped with two sample profilers: a general code profiler and a hotspots tracker. Both profilers make use of the CLR profiling API and reside in the Tool Developers Guide directory of the SDK. The general code profiler was adapted and used in the implementation of the Aspect-Oriented Programming system.

A .NET profiler runs as an in-process COM server. This means that the "client" and "server" run in the same process. The client in this case is the Common Language Runtime and the server is the profiler. An illustration of how this works is given in Figure 3.2.

Any COM-capable unmanaged language, can be used to implement the profiler. In this case C++ was used.

Each COM component is identified with a CLSID, which is a Globally Unique Identifier (GUID). A GUID is a pseudo-random number and can be generated automatically. The Microsoft .NET SDK provides a GUID generator, called `guidgen.exe` (see Figure 3.3).

[2]see Section 3.8 - Metadata & Reflection

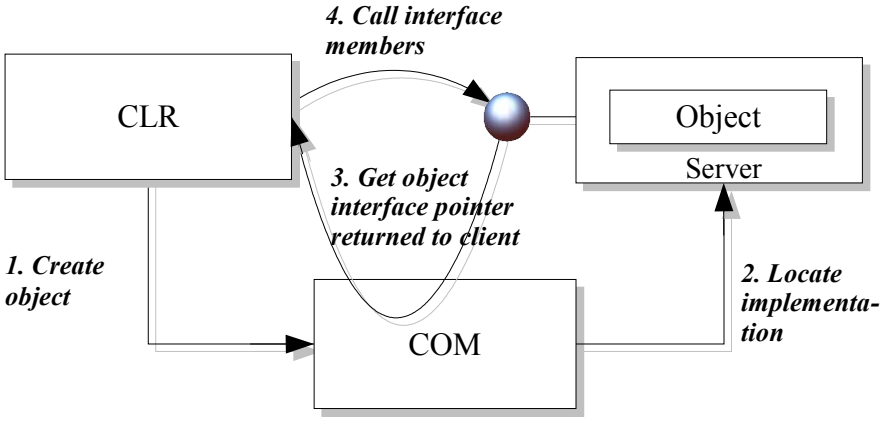

Figure 3.2: Object Creation In COM

This GUID is used as a registry value in the Windows registry. Applications that use a COM component will search the Windows registry for this identifier. To register a COM server in the Windows registry, the `regsvr32` command is used. The CLR will use this information to load the profiler into the managed application's address space.

3.8 Profiler Events

A .NET profiler must implement the `ICorProfilerCallback` interface. This interface provides a variety of methods that are triggered by the CLR. Any of these events[3], can be monitored by the profiler.

These events are focused on delivering information about an application's performance and memory usage. Mainly the provided events are loading, unloading and compilation events. The AOP system, presented in the next chapter is built around the `JITCompilationStarted` event.

[3] a list of monitorable events is included in the Appendix

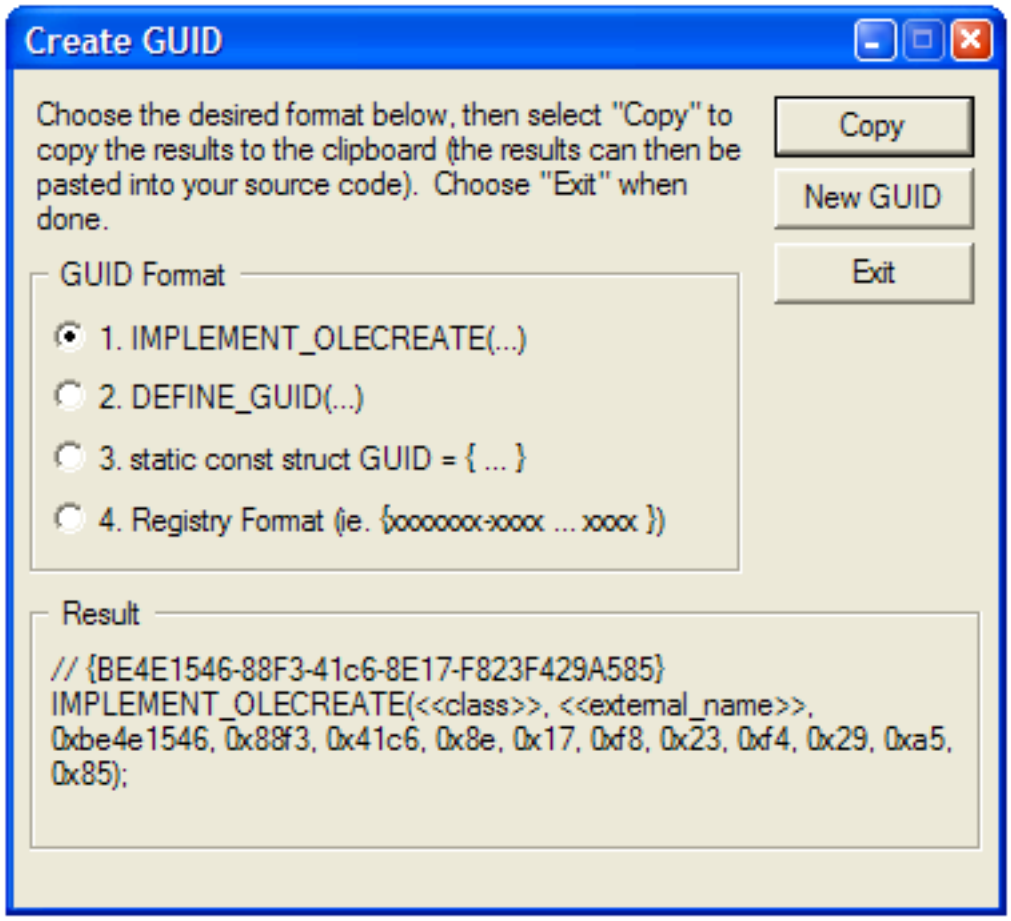

Figure 3.3: GUID

3.9 Metadata & Reflection

Metadata is data that describes other data. In the Microsoft .NET Framework metadata is used to describe the contents of an assembly. The fact that this metadata is stored within the assembly itself, gives it its self-describing ability.

The organization of metadata is done through the use of tables and heaps. Common Intermediate Language instructions reference this data via *metadata tokens* [34]. These tokens are 4-byte values. The first byte (most significant byte) indicates the type, the other 3 bytes either specify a row number in a table, or an offset in a heap. An example will help to gain a better insight in the usage of tokens.

```
public class HelloWorld
{
        public static void Main()
        {
                System.Console.WriteLine("Hello World!");
        }
}
```

Figure 3.4: Hello World In C#

The C# version of the famous HelloWorld program is shown in Figure 3.4. After this program is compiled, the resulting IL code can be viewed with, for instance the Microsoft .NET Framework IL Disassembler[4]. Figure 3.5 displays the corresponding Intermediate Language code.

Here all information is offered in a human readable format. The actual IL code in fact only contains opcodes and tokens. Opcodes used in this example are *call (0x72)*, *ldstr (0x28)* and *ret (0x2A)*.

[4] ildasm.exe /ALL HelloWorld.exe

```
.method /*06000001*/ public hidebysig static void  Main() cil managed
// SIG: 00 00 01
{
  .entrypoint
  // Method begins at RVA 0x2050
  // Code size       11 (0xb)
  .maxstack  1
  IL_0000:  /* 72   | (70)000001      */ ldstr      "Hello World!" /* 70000001 */
  IL_0005:  /* 28   | (0A)000002      */ call       void [mscorlib/* 23000001 */]
                                                    System.Console/* 01000003 */
                                                  ::WriteLine(string) /* 0A000002 */
  IL_000a:  /* 2A   |                 */ ret
} // end of method HelloWorld::Main
```

Figure 3.5: Hello World In IL

In addition, 5 different metadata tokens are used here:

- MethodDef (0x06000001)
- User string (0x70000001)
- AssemblyRef (0x23000001)
- TypeRef (0x01000003)
- MemberRef (0x0A000002)

There are two categories of metadata tokens, namely *definition* and *reference tokens*. Definition tokens define the entity, while reference tokens define references to the entity. For example, a MethodDef token describes a definition of a Method. A MethodRef token on the other hand describes how to find the MethodDef.

The layout of metadata (tables and heaps) can be viewed with a tool called metainfo[5]. Figure 3.6 displays the information obtained from the HelloWorld example with the help of metainfo.

From the 41 tables displayed by the metainfo tool, only the Method table is shown, which has sequence number 6. MethodDef tokens use this sequence

[5]in Tool Developers Guide\Samples\metainfo

6: Method Table

RVA				ImplFlags		Flags		Name	Signature	ParamList
00	00	20	50	00	00	00	96	**string#3b**	**blob#a**	Param[8000001]
00	00	20	68	00	00	18	86	string#40	blob#e	Param[8000001]

String Heap: 127(0x7f) bytes

00000001:	<Module>
0000000a:	HelloWorld.exe
00000019:	Mscorlib
00000022:	System
00000029:	Object
00000030:	HelloWorld
0000003b:	Main
00000040:	.ctor
00000046:	System.Diagnostics
00000059:	DebuggableAttribute
0000006d:	Console
00000075:	WriteLine

Blob Heap: 36(0x24) bytes

0,0 : > <
1,8 : b7 7a 5c 56 19 34 e0 89 > z\V 4 <
a,3 : 00 00 01 **> <**
e,3 : 20 00 01 > <
12,5 : 20 02 01 02 02 > <
18,4 : 00 01 01 0e > <
1d,6 : 01 00 00 01 00 00 > <

User Strings:

70000001 : (12) L"Hello World!"

Figure 3.6: Metadata Tables & Heaps

number to refer to the correct table in their most significant byte (see for instance Figure 3.5, which contained MethodDef token 0x**06**000001).

The Method table shown in Figure 3.6 has two entries. The first entry is the Main method, the second one is the default constructor method. The Main method will now be discussed in more detail.

Each column in the Method table is either 4 bytes or 2 bytes long. The first column contains the Relative Virtual Address (RVA), which for the main method is 0x2050. The second column specifies that this method is managed IL code. The Flags column in a logical OR of Public (= 0x6), Static (0x10) and Hide by signature (= 0x80). The Name column contains a pointer to an entry in the String heap (it points to the "Main" string).

Besides the String heap, there also exists a blob (Binary Large Object) heap and a User String heap. The blob heap is used to store items with variable length. For instance, the Signature of the Main method is stored here at position 0x0A.

The final column of the Method table holds pointers to the Param table. As can be seen from the entries in Figure 3.6, this table has sequence number 8 (0x**8**000001). Both entries point to the first entry in the Param table, which in this case is the NULL entry, because neither of the methods has parameters.

The .NET common language runtime provides two different APIs for reading and writing metadata [23]. The first of the APIs uses the .NET common language runtime and is called the Reflection API, while the second API is a set of unmanaged COM interfaces.

3.9.1 The Reflection API

The Reflection API allows access to metadata through classes in the `System.Reflection` namespace. The hierarchy of these classes is shown if Figure 3.7.

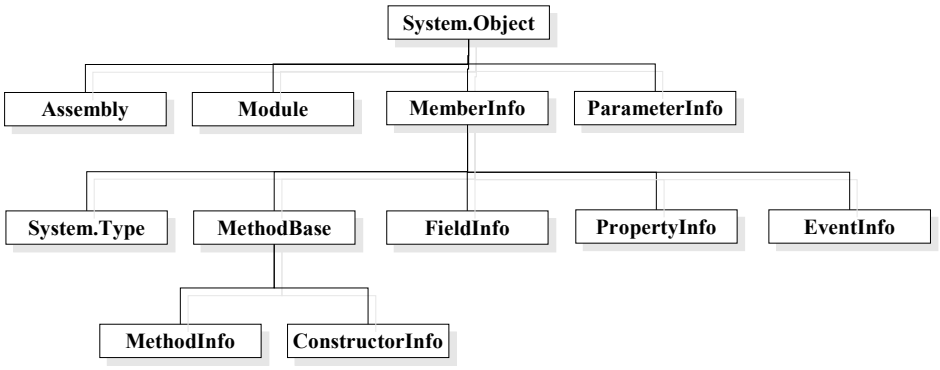

Figure 3.7: The Reflection Type Hierarchy

Of special importance here is the `System.Type` class, which encapsulates a representation of the type of an object. `Type` is the heart of reflection and represents type declarations (classes, interfaces, arrays, values and enumerations). Using the classes presented here allows to introspect existing types.

The `System.Reflection.Emit` namespace provides a variety of classes, that allow to build new types at runtime. Every element of a fully functional assembly can be created dynamically, right down to the Intermediate Lan-

guage instructions that make up method bodies.

The only drawback from using `System.Reflection.Emit` is that the modification of existing types is not permitted. This limitation makes a dynamic AOP solution based on `System.Reflection.Emit` a lot less flexible. It would be a lot easier to start from an existing type and make a few necessary adjustments, than to create entirely new types to use as wrappers. An alternative for the Reflection API, called the Unmanaged Metadata API, provides a solution here.

3.9.2 The Unmanaged Metadata API

The Unmanaged Metadata API comprises a set of unmanaged COM classes, which provide more functionality. In addition to the operations provided by the Reflection API, it allows the manipulation of existing types. More specifically, the emission of metadata is provided by the IMetaDataEmit interface. The import interface is IMetaDataImport. This API requires to be used in an unmanaged coding environment, such as unmanaged C++. Also some basic knowledge of COM is needed.

The COM object that needs to be loaded into memory is the IMetaDataDispenser object. Invoking the IMetaDataDispenser::OpenScope method on an existing assembly, maps its metadata into memory. This in-memory copy can then be queried with the IMetaDataImport interface or added-to with the IMetaDataEmit interface.

The methods of the IMetaDataEmit interface enable additions to the metadata tables, as well as additions to the String, blob and User String heaps. For instance the DefineUserString method stores a user string into the User String heap. This method will also return a token, denoting the offset in the heap. Similarly, DefineMethod will add a new row to the Method table. The input parameters are the values for every column, the output parameter is the method token.

3.10 Attributes

The Common Language Runtime allows to extend metadata with user defined descriptive attributes, called *custom attributes* [5] [2]. This is a very powerful feature that allows information about types or methods to be put directly inside the assembly, alongside the relevant code.

Attributes are specified between square brackets in C#. An example is shown in Figure 3.8.

```
using System;
using Aspects;


public class LoggerAspect
{

        [Alias("Simple.method1")]
        public void Before__m1()
        {
                Console.WriteLine("Before second");
        }

}
```

Figure 3.8: Example with Attributes

These attributes can be placed on top of fields, methods, classes and even assemblies. In the example here, a new attribute is declared on top of a method. Notice that every attribute has to be derived from the `System.Attribute` class. An attribute is thus an ordinary class, whose constructor is used to describe other elements of an assembly.

Because attributes are part of an assembly's metadata, they can be queried and created at runtime using reflection. For Aspect-Oriented Programming purposes, these attributes are useful to specify pointcuts. The Aspect Engine presented in the next chapter utilizes attributes in this manner. The definition of AliasAttribute used in Figure 3.8 is shown in Figure 3.9.

All custom attributes of an assembly can be retrieved with a call to `System.Type.GetCustomAttributes`. In the example above, this would re-

```
using System;

namespace Aspects
{
        [AttributeUsage(AttributeTargets.Method)]
        public class AliasAttribute : Attribute
        {
           private string myName;

           public AliasAttribute(string name)
           {
              myName = name;
           }

           public string Name
           {
              get { return myName; }
           }
        }
}
```

Figure 3.9: Attribute used to define a pointcut

turn an object of type AliasAttribute. The poincut, passed on in the constructor of this AliasAttribute, can be retrieved through the Name property.

A special attribute is the AttributeUsage attribute. This attribute is placed on top of an attribute class and indicates what the permitted usages are. Here it demands that the AliasAttribute is used to decorate methods only.

Chapter 4

A Different Approach

Existing implementations of dynamic Aspect-Oriented Programming, developed for Java, often make use of the Java Virtual Machine Debugger Interface (JVMDI). A logic step toward dynamic AOP support for the .NET platform would be to look into the Microsoft CLR debugger interface.

Although this is an obvious step toward a proof of concept, it is intuitively clear that the use of a debugger causes a significant performance penalty. This can be accounted to the fact that a lot more administration is done for each method and class. Also, previous Java implementations have illustrated this fact [25] [24] [4].

An alternative approach is the use of the Reflection API, which has already been shown by the LOOM.NET [29] implementation. However, this implementation also demonstrated that the use of the Reflection API is very limited and does not provide the desired dynamism and flexibility.

In trying to avoid these demotivating approaches, the Profiler API looked the most appealing. This API allows monitoring of more fine-grained events than the debugger. In addition, it permits to monitor just a minimal set of events, to reduce event notification overhead.

This section will give an in-depth overview of a dynamic Aspect-Oriented Programming system for the Microsoft .NET platform, making use of the CLR Profiling API. For sake of simplicity, this system is referred to as *Aspect Engine* throughout the remainder of this document.

4.1 Prerequisites

The system proposed here requires a working version of the Microsoft .NET Runtime to be installed. It has been successfully tested on v1.0.3705 and v1.1.4322 of the Microsoft .NET Framework. Because this AOP system uses the CLR profiler, other implementations of the Common Language Infrastructure will not be able to use it. For instance Mono[1], an open source implementation of the CLI, has its own profiling API, which renders this tool inapplicable for applications running on Mono.

4.2 Front-end

A Graphical User Interface (GUI) is provided to visualize all available aspects (see Figure 4.1).

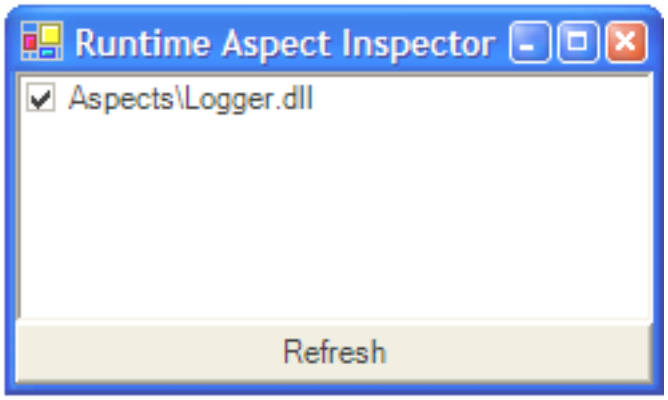

Figure 4.1: Graphical User Interface

When the Refresh button is pressed the Aspect Engine will search for new aspects in the 'Aspects' subdirectory. If one or more aspects are found, they are activated immediately and displayed in the GUI. Every Aspect is provided with a checkbox to enable or disable it. In Figure 4.1, the only aspect found was Logger.dll.

[1]http://www.go-mono.com

4.3 Architecture

Figure 4.2 shows the organization of the implemented AOP system.

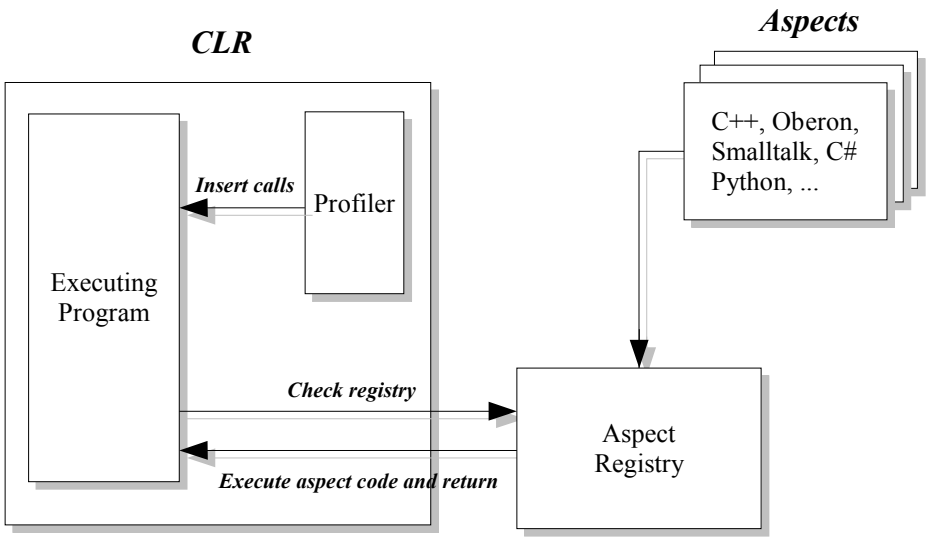

Figure 4.2: AOP System Architecture

The architecture of the AOP system is divided into two different parts: a profiler part and a registry part. The Aspect Registry contains all information about available aspects. These aspects can be written in C#, Oberon.NET, C++ or any .NET compatible language.

The task of the profiler is to insert calls to the Aspect Registry. In fact, the profiler has to insert 3 different calls into the running application:

- A call to startup the registry (before the main method, needed for initialization)
- A call to check if there exists any before-advice (before every method)
- A call to check if there exists any after-advice (after every method)

Figure 4.3 depicts the result of these additions.

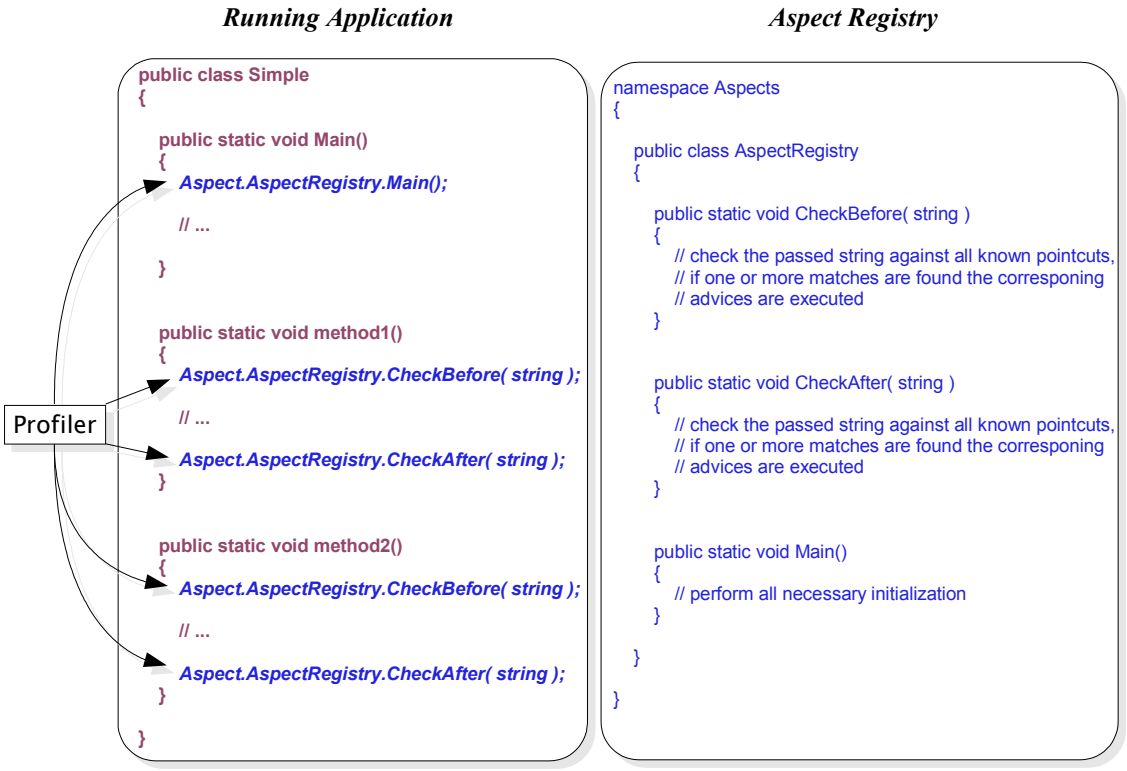

Figure 4.3: Profiler & Aspect Registry

Every method will consistently start and exit with a CheckBefore and CheckAfter method respectively. Figure 4.3 is in fact a high level view of the resulting application code, because the real modifications will be performed at the Intermediate Language level.

So to sum up what happens when an application is executed, with the Aspect Engine enabled, a list of consecutive actions is given below:

- First the profiler will be loaded into memory
- The application code will be augmented with several calls to the Aspect Registry
- As a result of these additions, the application itself will check the registry, to see if the currently executing method has matching pointcuts
- For every match found, the Aspect Registry will execute the corresponding advice
- The Graphical User Interface offers an easy way to switch available aspects on or off, by clicking the corresponding checkbox
- Hitting the refresh button will force an update of the Aspect Registry

4.4 Chronological Overview Of The System

Next an elaborate chronological overview of the system is given, from program startup to addition and removal of aspects.

4.4.1 Loading the Profiler

When a managed application is executed, the Common Language Runtime looks for the `COR_ENABLE_PROFILING` environment variable. If this is set to 1, the CLR will look for a profiler CLSID in the Windows registry. The CLSID to look for is provided by the `COR_PROFILER` variable.

In short the following commands have to be executed to enable profiling support:

- `set COR_ENABLE_PROFILING = 0x1`
- `regsvr32 ProfilerGCP.dll`
- `set COR_PROFILER = {01568439-E2BA-4434-8ACC-816239E8B8B5}`

These commands can be executed in the Windows Command Prompt. The `regsvr32` command registers the ProfilerGCP dll, with generated GUID = 01568439-E2BA-4434-8ACC-816239E8B8B5, as a COM server. The resulting entry in the Windows registry is shown in Figure 4.4, using `regedit.exe`.

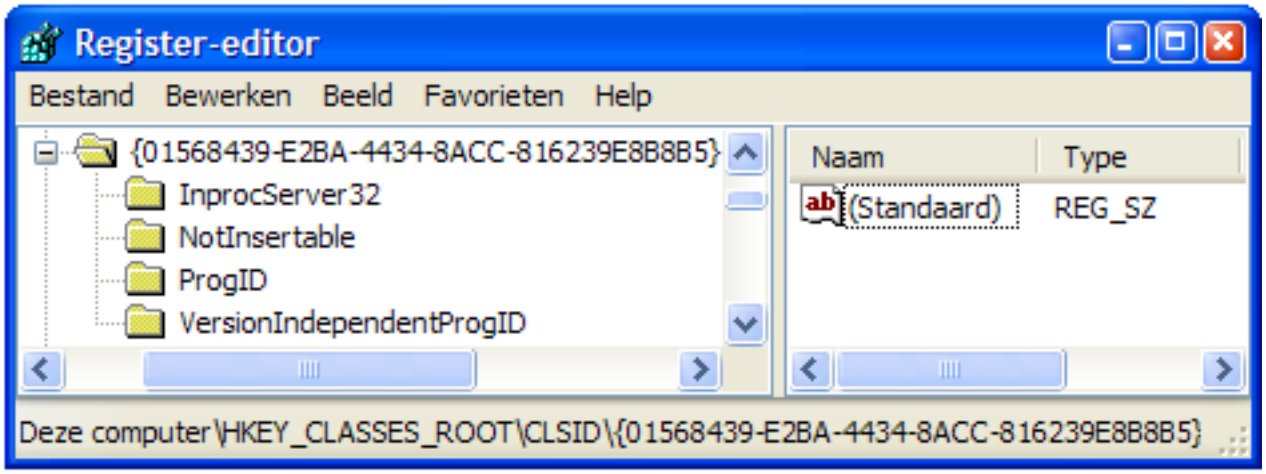

Figure 4.4: The Windows Registry

4.4.2 Setting The Profiler Bitmask

To set options for the profiler, a 32-bit bitmask is used. Each bit in this bitmask operates as a flag to indicate whether a certain option should be enabled or disabled. The Aspect Engine's profiler is set to monitor module loading events, JIT compilation events and to disable the use of inlining. This results in the bitmask shown in Figure 4.5.

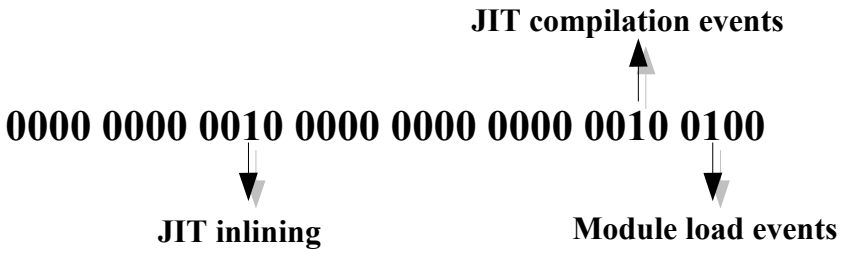

Figure 4.5: The Profiler Bitmask

Monitoring the Just-in-time compilation events allows to observe when the JIT compiler is activated. More specifically, the `JITCompilationStarted` event is fired by the CLR whenever a method is executed for the first time (otherwise the cached version of the code is used[2]). When this happens, control is passed to the profiler and the currently executing program is suspended between method entry and execution of the method body. From an AOP perspective these intervention points can be seen as join-points.

Even more interesting is the fact that, inside the `JITCompilationStarted` event handler the Intermediate Language code of the not-yet JIT compiled method is available. The IL code is then also subject to possible modifications. Instructions can be added or removed and the entire method's body can even be replaced with other code (if the signature of the method remains unaltered).

Another flag in the bitmask is used to disable inlining. Inlining is frequently used to replace calls to static methods with their method body. This introduces additional difficulties when altering Intermediate Language code, so it has to be turned off.

[2]see Chapter 3 - 3.3 JIT Compilation

4.4.3 Profiler startup

After the CLR has successfully loaded the profiler in the application's address space, it calls the profiler's Initialize method. The signature of this method is:

```
HRESULT Initialize( IUnknown * )
```

The ICorProfilerInfo interface can be obtained from the IUnknown interface by calling IUnknown::QueryInterface. Once this interface is retrieved it can be used to set the event bitmask. Later, it will be used to gain access to metadata and to retrieve and insert Intermediate Language code. To be able to use the ICorProfilerInfo interface in the future, it is stored as a member field of the profiler.

The remaining task of the profiler is to provide every method of the running application with calls to the Aspect Registry. To call the Aspect Registry's methods, the tokens of these methods should be retrieved. This is where the Unmanaged Metadata API[3] comes in.

As seen before, this API consists of a set of COM interfaces. From these interfaces, the IMetaDataImport interface is needed, which provides read-only access to the metadata tables of assemblies.

The first table that is checked is the TypeDef table, which contains information about all types in the Aspect Registry dll. Among this information is the type's name, namespace name and a list of its MethodDef tokens. The needed entry is the one that has 'Aspect Registry' as type name and 'Aspects' as namespace name. When this entry is identified, the list of associated MethodDef tokens is retrieved.

As explained in section 3.9 – MetaData & Reflection, MethodDef tokens identify an entry in the Method table. Enumerating all entries in this table returns a list of method signatures and corresponding MethodDef tokens. The signatures are used to verify which of the obtained tokens belongs to the CheckBefore, CheckAfter and Main method of the Aspect Registry class.

[3]see Chapter 3 - 3.9.2 The Unmanaged Metadata API

Since the method tokens inside the Aspect Registry dll remain unchanged, it is only necessary to retrieve them once. Therefore, the code to resolve these tokens is put inside the Initialize method. Once the initialization is complete, the profiler waits for events to be fired by the Common Language Runtime.

4.4.4 Inserting Method Calls

Every method that is executed for the first time will be passed to the JIT compiler, each time firing a `JITCompilationStarted` event. The handler for this event is implemented by the profiler and has the following signature:

```
HRESULT JITCompilationStarted( FunctionID, BOOL )
```

The FunctionID identifies the method that fired the event. The second parameter indicates whether or not it is safe to block this function. Within the event handler, the method calls to the Aspect Registry will be added.

Extra caution is to be taken to ensure that the Aspect Registry methods themselves do not get involved in the process. This would lead to infinite loops, because the Aspect Registry methods will contain calls to themselves. For sake of simplicity, the methods of the System namespace are also left out of the process. Most of the time the internal use of these methods is transparent for the programmer and not intended to be targeted by aspects.

Now that only the running application's methods are distilled, the InsertCallToRegistry method is called. This method has the following signature:

```
HRESULT InsertCallToRegistry( FunctionID, BOOL )
```

The first parameter is the same FunctionID that was passed to the JITCompilationStarted event handler. It will be used to obtain all necessary information. The second parameter that is passed on will be true if the method under consideration is the Main method of the running application and false otherwise.

Inside this method the ICorProfilerInfo::GetFunctionInfo method is called, which delivers the following information:

- ClassID, the method's parent class
- ModuleID, the module in which the method is defined
- MethodToken, the metadata token for the method

After this, the ModuleID is passed on to the ICorProfilerInfo::GetModuleMetadata method, which returns the IMetaDataEmit interface. In turn, the IMetaDataAssemblyEmit interface can be obtained from this interface by calling QueryInterface on it.

The IMetadataAssemblyEmit interface is again part of the Unmanaged Metadata API and allows to insert metadata into existing assemblies (remember that this is not possible with the Reflection API). Using these two interfaces the following additions to the running application's metadata are made:

- A reference to AspectRegistry.dll is made, by adding an entry to the AssemblyRef table
- A reference to the Aspects.AspectRegistry class is made, by adding an entry to the TypeRef table
- Three references are added to the MemberRef table: one for AspectRegistry::Main, one for AspectRegistry::CheckBefore and one for AspectRegistry::CheckAfter

The running application now knows of the existence of the Aspect Registry and can successfully resolve the following of its methods:

```
public static void Main()
public static void CheckBefore(string signature)
public static void CheckAfter(string signature)
```

The argument that will be passed on to the CheckBefore and CheckAfter methods, is the signature of the method that invokes them. This will be used by the registry to check for matching join-points.

As with methods, strings are coded with a token. To obtain such a token, the IMetaDataEmit::DefineUserString method has to be called. This method will add the custom made string to the User String heap and return a string token. This string token will then represent the offset of the custom string in the User String heap.

The preparation of metadata is now complete and it is time to add the Intermediate Language instructions that implement the actual method calls. The IL instructions to pull this off are [19]:

```
ldstr <User-defined string token>
call <MethodRef token>
```

In binary notation the ldstr and call instructions are each one byte long and have opcodes 0x72 and 0x28 respectively. Tokens are 32 bits long, resulting in a total of ten bytes of IL code.

To retrieve the Intermediate Language code of a method, the ICorPorfilerInfo::GetILFunctionBody method is used. The signature of this method is given below.

```
HRESULT GetILFunctionBody( ModuleID, mdMethodDef, LPCBYTE **, ULONG64 * )
```

The metadata token for the method, as well as the moduleID are required input parameters. The GetILFunctionBody method returns a pointer to the body of the given method, starting at its header [26]. An illustration of the returned information is given in Figure 4.6 (information taken from [22]).

Besides the Intermediate Language code, the only field of interest here is the Size field. This field contains the size of the actual IL code, without the method header. When modifying the IL code, this field has to be adjusted accordingly.

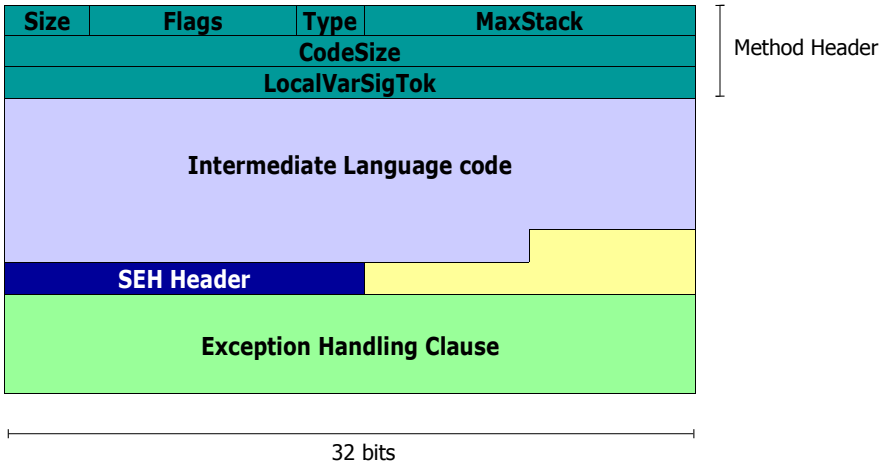

Figure 4.6: IL Method Body Layout

Altering the Intermediate Language code, returned by `GetILFunctionBody`, will result in an access violation, because it is read-only. To circumvent this, a copy of the image is made. Necessary changes are then performed on this copy. Afterwards, the original method body can be replaced with the modified version, using `SetILFunctionBody`.

4.4.5 Calling the Aspect Registry

The profiler's task stops at adding registry calls to the running application. From this moment on, the Aspect Registry will start receiving method invocations. The first of these will invoke the Aspect Registry's Main method. The purpose of this call is threefold:

- It will initialize the tables holding information about available aspects
- A first check of the Aspects subdirectory is made, to look for aspect assemblies
- The Graphical User Interface is started

It is not necessary to constantly poll the Aspects subdirectory to verify the existence of new aspects. Due to the infrequent addition of aspects, it is better to perform an update check explicitly. This can be done by pressing the Refresh button in the GUI.

Newly found aspect assemblies are immediately analyzed by the Aspect Registry. It will then, load them into memory, extract all required information and organize it in tables. One of these tables is used to store the aspects that are currently active. The entries in this activity table are modified by toggling the checkboxes, displayed in the Graphical User Interface. This table contains only the names of applied aspects. More detailed tables are used to store properties of aspects.

Separate tables are used for before and after advice. Each of these contain mappings of every pointcut to one or more advices. The CheckBefore and CheckAfter method, when executed, contain the signature of the method that invoked them. This signature can then be used to check the Before and After tables, to search for matching pointcuts. The content of the CheckBefore method is discussed next.

When the CheckBefore method finds a matching poincut, it will determine the aspect(s) this poincut belongs to. For each aspect it will then consult the activity table. If an entry for the aspect is present this means that the aspect should be applied.

In the Before table, each pointcut is mapped to a list of its applicable advices. For each pointcut description that matches the method's signature, this list is retrieved. The advices contained in the list are then subsequently executed.

The CheckAfter method works completely analogous, except that it consults the After table, instead of the Before table.

4.5 Performance

Although the developed AOP system is not tuned for optimal performance, it refrained from using for instance the debugger, because of its known performance overhead. Therefore, it is still interesting to see the results of this alternative approach. The benchmark used to test the impact of a separate aspect registry is explained below.

4.5.1 Testbed & benchmark setup

Figure 4.7 displays the testbed architecture used in the benchmark.

CPU	2.66Ghz P4
Motherboard	Micro-Star International Co., LTD MS-6701
Chipset	Silicon Integrated Systems (SiS) SiS648
Memory	768Mb (DDR333)
Graphics	GeForce4 Ti 4200
Operating System	Microsoft Windows XP Home Edition Service Pack 1 installed

Figure 4.7: Testbed

The execution time of an application, making use of the Aspect Engine, is compared to the execution time of the same application that already contains aspects. This allows to monitor the additional overhead incurred by consulting the registry.

Figure 4.8 presents a test application. The current date and time are checked before and after the test. Calculating the difference between these two points gives the time needed to execute a method, together with all associated advice. One version of the test program already contains the advice, while in the second version advices will be obtained at runtime from the Aspect Registry.

As can be seen in Figure 4.8, the tested method is already executed once, ahead of actual measurements. This is to avoid the JIT compilation phase, which will only affect the first invocation of a method.

```
public class App
{
   public static void Main()
   {
      Simple s = new Simple();

      s.second();

      long start =  System.DateTime.Now.Ticks / 10000;

      for (int i = 0; i < 10000; ++i)
      {
         // before advice
         s.second();
         // after advice
      }

      long end = System.DateTime.Now.Ticks / 10000;

      System.Console.WriteLine(end - start);
   }

   public void second()
   {
      // ...
   }
}
```

Figure 4.8: Benchmark application

The first application, containing the advices, was timed at 2922 milliseconds. The application using the Aspect Engine was timed at 3188 ms. This implies a 266 millisecond slowdown, or in other words a 9% increase in execution time.

Reasons for this delay are quite easily identified. Every method contains two extra calls to the Aspect Registry. Also, each of these calls searches the registry for available aspects and matching pointcuts. Although these calls provide the needed flexibility to easily add and remove aspects to the running application, they do require some extra time.

Inserting the advices themselves into the running application would most likely increase performance. However, this would destroy the existence of aspects at runtime, making it a lot harder to remove them later.

4.6 Other implementations

	AspectJ	Aspect C#	RAP. LOOM.NET	Aspect Engine
Supported Languages	Java	C#	Any .NET compatible language	Any .NET compatible language
Addition / removal of aspects	Requires recompilation and restart of the application	Requires recompilation and restart of the application	Requires recompilation and restart of the application	No recompilation or restart required
Availability of source code	Required	Required	Required	Not required
Runtime overhead	0 %	0 %	Slower application startup	Slower application startup, 9 % method invocation latency
Language extensions	Yes	No, uses XML files	No, uses XML files	No
Currently available language constructs	Before, after, around, cflow...	Before, after, around	Before, after, around	Before, after

Figure 4.9: Other implementations

Figure 4.9 summarizes some characteristics of other AOP implementations, compared to the Aspect Engine presented in this chapter.

Although AspectJ is an implementation for the Java language, it is included in the listing because it represents the unofficial standard for all Aspect-Oriented languages. It provides the widest range of features and language contructs. Aspect C#, LOOM.NET and the Aspect Engine all implement a subset of these.

Again the latency caused by using a dynamic approach is emphasized. The absence of the method invocation latency in the RAPIER LOOM.NET approach can easily be explained. Weaving the application code and aspects at runtime will lead to slower application startup, but after the weaving process is complete, the weaved method will show no additional latency compared to the same method that would contain advices before application startup.

4.7 Limitations

The prototype implementation of the Aspect Engine presented in this chapter was tested on small scale applications. The results show that it is possible to implement a logging aspect that targets multiple methods in a variety of different classes.

The current functionality is limited, as it for instance only allows *method execution join-points*. Figure 4.10 lists the type of join-points available in AspectJ.

Join-point	Point in program execution
method call	method or constructor of a class is called
method execution	an object receives a method or constructor call
field get or set	a field of an object is accessed
exception handling execution	exception handler invoked
class initialisation	any static initialiser are run.
object initialisation	any static initialiser are run during object creation

Figure 4.10: Available join-points in AspectJ

There is also room for improvement in the way aspects are defined. A first improvement is the use of wildcards. Wildcards are used in AspectJ to facilitate the specification of pointcuts. In the current version of the Aspect Engine, pointcuts have to be defined by means of regular expressions. Introducing these wildcards, using regular expressions behind the scenes, would greatly simplify the definitions of pointcuts.

Another possible improvement is to force a more controlled use of objects targeted by aspects. For now, aspects can access any kind of object, using reflection. A solution is to provide a wrapper object as in the RAPIER LOOM.NET approach, which hides the details of reflection and forces to use only the object targeted by the aspect.

Chapter 5

Conclusion & future work

5.1 Conclusion

The implementation of a dynamic Aspect Oriented Programming system, presented in the previous chapter, proves it is possible to make use of AOP in the .NET environment.

Developing this solution did not require changes to the internals of the Microsoft .NET platform itself. Instead it employs a set of COM interfaces that query the Common Language Runtime and allow to modify applications running inside it.

Applications are adapted is such a way that they will consult a registry to check for available aspects. Doing so leads to a performance slowdown, but at the same time introduces the needed flexibility. Augmenting the program code with calls to a central registry, as opposed to inserting aspect advice directly into the running application, makes removing aspects a lot easier. This is due to the fact that aspects continue to exist at runtime.

Aspects can be fully defined in any programming language providing a .NET compiler, without the use of new keywords or constructs. Pointcuts are specified by means of attributes, which are placed on top of methods. To further refine when these methods are to be executed, a prefix to the name of the aspect is used. This prefix will mark the method as being either before or after advice.

The implication of using the `JITCompilationStarted` event, provided by the Profiler API, as point of interaction, is a restriction on the AOP constructs that can be implemented. For instance, AspectJ distinguishes 2 types of join-points: *method call join-points* and *method execution join-points*.

As the names imply, the first type of join-point identifies a point in a program when a method is called, while the second type identifies a point at which the body of code for an actual method executes. The implemented AOP system only allows to specify execution join-points, because the `JITCompilationStarted` event is thrown at this exact moment.

Other approaches to implement dynamic AOP for the Microsoft .NET platform are likely to be possible. The runtime overhead created by these approaches might well outperform the implementation chosen here. However, one of the choices in this thesis was to avoid internal modifications of the .NET Framework. Other implementations may not refrain from doing so and achieve better performance in this manner. Also, future versions of the .NET Framework could offer more implementation options and facilitate the construction of an aspect system.

5.2 Future work

Before and after advice can be implemented cleanly using the `JITCompilationStarted` event. At that point, the profiler can safely access the Intermediate Language code of the method that caused this event to be thrown. Performing a detailed analysis of the obtained IL code, should lead to the implementation of more complicated AOP constructs.

For instance, call join-points can be identified at the Intermediate Language level. In fact, all information that is available to a static aspect weaver is at some point available to the profiler, in combination with the unmanaged metadata API. The feasibility of extending the AOP system with additional constructs and the hereby induced overhead form a basis for interesting further study.

Appendix A

Events available to the profiler

- Application domain creation and shutdown events
- Assembly load and unload events
- Module load and unload events
- COM-callable wrapper creation and destruction events
- Just-in-time (JIT) compilation events
- Class load and unload events
- Thread creation and destruction events
- Function enter and leave events
- Object allocation and garbage collection events
- Transitions between managed and unmanaged code events
- Context and remote boundary crossing events
- CLR suspend and resume events
- CLR (managed) exception events

Bibliography

[1] http://aosd.net, 2003. Aspect Oriented Software Development website.

[2] Tom Archer. *Inside C#*. Microsoft Press, 2001.

[3] http://eclipse.org/aspectj/, 2003. AspectJ website.

[4] Swen Aussmann and Michael Haupt. Dynamic aop through runtime inspection and monitoring. Technical report, Darmstadt University of Technology, 2003. First workshop on advancing the state-of-the-art in Run-time inspection.

[5] Don Box and Chris Sells. *Essential .NET, Volume 1: The Common Language Runtime*. Addison Wesley, 2002.

[6] Frederick P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[7] Yoonsik Cheon, Yoshiki Hayashi, and Gary T. Leavens. A thought on specification reflection. Technical Report 03-16, Department of Computer Science, Iowa State University, Dec 2003.

[8] Shigeru Chiba. Load-time structural reflection in Java. *Lecture Notes in Computer Science*, 1850:313–??, 2000.

[9] Allstair A. R. Cockburn. The impact of objectorientation on application development. *IBM Systems Journal vol 32 no 3*, 1993.

[10] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods Tools and Applications*. Addison-Wesley, 2000.

[11] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[12] Andreas Gal, Wolfgang Schrder-Preikschat, and Olaf Spinczyk. Aspectc++: Language proposal and prototype implementation. Technical report, University of Magdeburg Germany, 2001.

[13] Wasif Gilani and Olaf Spinczyk. A family of aspect dynamic weavers, 2004. AOSD 2004 - Dynamic Aspects Workshop Lancaster UK.

[14] Robert Hirschfeld. Aspects aspect-oriented programming with squeak. Technical report, DoCoMo Communications Laboratories Europe, 2003.

[15] ECMA International. Common language infrastructure. In *Standard Ecma-335 - 2nd Edition*, 2002.

[16] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT press, 1991.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[18] Howard Kim. Aspectc#: An aosd implementation for c#. Master's thesis, Trinity College, Dublin, 2002.

[19] Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[20] Logging in org.apache.tomcat, 2002. Tutorial: Aspect-Oriented Programming with AspectJ (1.0.6), Powerpoint presentation, shown at OOPSLA 2002.

[21] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pieere Verbaeten. Aspects should not die, 1997. Position paper at the ECOOP 97 workshop on Aspect-Oriented Programming.

[22] Aleksandr Mikunov. Rewrite msil code on the fly with the .net framework profiling api. *MSDN Magazine*, 2003.

[23] Matt Pietrek. Avoiding dll hell: Introducing application metadata in the microsoft .net framework. *MSDN Magazine*, 2000.

[24] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109. ACM Press, 2003.

[25] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002.

[26] Profiler documentation, 2003. Microsoft .NET v1.1 Tool Developers Guide docs.

[27] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *Proceedings of the second international conference on Generative programming and component engineering*, pages 189–208. Springer-Verlag New York, Inc., 2003.

[28] Wolfgang Schult and Andreas Polze. Aspect-oriented programming with c# and .net. Technical report, Hasso-Plattner-Institute at University Potsdam, 2002.

[29] Wolfgang Schult and Andreas Polze. Dynamic aspect-weaving with .net, 2002. Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen.

[30] Bjarne Stroustrup. *The C++ Programming Language, Thirth Edition.* Addison-Wesley, 2000.

[31] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[32] Julian Templeman and John Paul Mueller. *COM Programming with Microsoft .NET.* Microsoft Press, 2003.

[33] http://jakarta.apache.org/tomcat/, 2003. The Jakarta Site - Apache Tomcat.

[34] Unmanaged metadata documentation, 2003. Microsoft .NET v1.1 Tool Developers Guide docs.

[35] John W. Verity and Evan I. Schwartz. Software made simple. *Business Week*, 1991. p. 92-97.