Aspect.NET

A cross-language aspect weaver

Bjørn D. Rasmussen, Casper S. Jensen,
Jimmy Nielsen & Lasse Jensen

June, 2004

Aalborg University
Department of Computer Science· Fredrik Bajers Vej 7E
9220 Aalborg Øst

The Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE: Aspect.NET - A cross-language aspect weaver

PERIOD:
2004-02-02 – 2004-06-11

GROUP:
Bjørn D. Rasmussen
Casper S. Jensen
Jimmy Nielsen
Lasse Jensen

SUPERVISOR:
Bent Thomsen

NUMBER OF COPIES: 8

NUMBER OF PAGES: 125

ABSTRACT:

Aspect-oriented programming is designed to compensate for traditional languages' inability to encapsulate what is referred to as crosscutting concerns. Typical examples of crosscutting concerns are things like logging, debugging, synchronization, etc. On the Java platform the aspect-oriented extension for the Java language, AspectJ, has already shown to provide developers with a simple way of expressing crosscutting concerns in Java applications. Crosscutting concerns, however, pose a problem to all languages which have a key abstraction and composition mechanism that is rooted in some form of generalized-procedure. This report therefore investigates the feasibility of designing a cross-language aspect-oriented extension for the Common Language Infrastructure (in particular the .NET Framework) that will allow aspect weaving regardless of language.

Det naturvidenskabelige fakultet
Aalborg Universitet

Institut for datalogi

TITEL: Aspect.NET - A cross-language aspect weaver

PERIODE:
2004-02-02 – 2004-06-11

GRUPPE:
Bjørn D. Rasmussen
Casper S. Jensen
Jimmy Nielsen
Lasse Jensen

VEJLEDER:
Bent Thomsen

ANTAL KOPIER: 8

ANTAL SIDER: 125

SYNOPSIS:

Aspekt-orienteret programmering er designet til at kompensere for traditionelle programmeringssprogs manglende evne til at indkapsle, hvad der normalt bliver refereret til som "crosscutting concerns". Typiske eksempler på "crosscutting concerns" er logging, debugging, synkronisering osv. På Java platformen, har den aspekt-orienterede udvidelse til Java sproget, AspectJ, vist sig at forsyne udviklere med en simpel metode til at udtrykke "crosscutting concerns" i Java applikationer. "Crosscutting concerns", udgør imidlertid et problem for alle sprog, som har en abstraktion og kompositionsmekanisme, der stammer fra en generaliseret procedure. Derfor undersøger denne rapport muligheden for at designe en aspekt-orienteret udvidelse til "Common Language Infrastructure" på tværs af programmeringssprog (Mere præcist til .NET platformen) som tillader indfletning af aspekter uanset sprog.

Summary

A software system is often an implementation of multiple concerns which can roughly be put into two categories, namely business logic concerns and technical concerns. Business logic concerns are typically the core functionality in a software system whereas the technical concerns are functionality that is typically imposed by developers. Traditional programming languages, and object-oriented languages in particular, provide excellent support for modularizing business logic concerns, but technical concerns, on the other hand, often pose a problem because they are not easily modularized in traditional languages. This observation has lead to development of aspect-oriented programming (AOP) which has been designed to compensate for traditional languages lack of support for modularizing concerns that tend to cut across several modules. Such concerns are often termed crosscutting concerns.

Due to the varieties that exist in programming languages it is, however, not possible to build a common AOP extension for all languages. The new .NET platform, however, include a common language runtime which is designed to provide a single runtime system for all languages. Many of the traditional languages are being ported to this new platform and since all languages for this platform compile to a common format, namely the common intermediate language, it is then possible to support aspect weaving for all languages by weaving aspect into their common format. The goal for this master thesis has therefore been to provide .NET developers with an aspect weaver that will allow users to weave aspects into their applications regardless of the language they are using. The aspect weaver being developed in this project supports aspect weaving by providing users with a programming language which to a great extend has been influenced by the design of the dominant AOP-language for the Java platform, namely AspectJ, and for this reason the language being developed in this thesis is called Aspect.NET. Supporting aspect weaving for all .NET languages is, however, quite a challenge and therefore the initial design of Aspect.NET only investigates how a limited number of languages can be supported for aspect weaving. These languages are the following: C#, J#, JScript, Managed C++, C++, Visual Basic.NET, Eiffel.NET, SML.NET, F# and Component Pascal.

The first part of this thesis gives a detailed description of the .NET framework and in particular the core components which makes it possible for .NET to provide a common runtime system for all programming languages. These core

components are the Common Type System which are the types that are shared among languages, the Common Language Runtime which is .NET's virtual execution system, the Common Intermediate Language which is the language that all .NET languages must be translated to, and finally metadata is introduced which is used to describe the compiled code. Next AOP is introduced and how it will assist traditional languages in modularizing concerns which would normally be scattered across several modules. Currently aspect weavers supports two ways of weaving aspects into programs, namely either at compile-time or at runtime. Following the introduction of AOP are listed the current technologies which are able to weave aspect into programs and how they perform this weaving operation.

Once the preliminary theory is set the second part outlines how a cross-language aspect weaver can be designed. The most important concept when designing a cross-language aspect weaver is being able to locate calls to methods which represent crosscutting concerns and two approaches for allowing this are introduced. The approach that will be used for the design of Aspect.NET is then selected along with a discussion of the advantages and disadvantages of the selected approach. The syntax and semantics of the Aspect.NET programming language are also outlined in the second part, which shows how users are able to capture and implement crosscutting concerns in their preferred programming language.

The third part outlines the overall implementation of the Aspect.NET compiler and gives a detailed description of how each of the more advanced language constructs are implemented. Examples of how to use the Aspect.NET languages and compiler are outlined in the fourth part. These examples include a simple "Hello world" application, an example of optimization by mixing languages and also an example of how Aspect.NET can be used to solve synchronization issues.

The fifth part presents the results which have been achieved during the development of Aspect.NET. The key points in these results are that the Aspect.NET language has been design as an aspect-oriented programming language for CIL because code-mangling during compilers' translation to CIL makes it very difficult to provide a more high-level approach for capturing aspect in .NET languages. This is mainly due to the fact that CIL is based on a quite restricted object-oriented model for expressing programs, which makes languages from e.g. the functional programming paradigm very difficult to express in CIL.

Preface

This report is a master thesis in computer science and has been prepared at the Department of Computer Science at Aalborg University, Denmark, during the period from the February 2nd to June 11th, 2004.

This master thesis is prepared under the research unit focusing on database and programming technologies. The main focus of this master thesis is on applying aspect-oriented programming to languages for the .NET Framework. This is attempted by using the Common Intermediate Language such that aspects can be applied regardless of the language used on the .NET platform.

This thesis has been written by project group E4-114 as joint work, and accordingly the work has been divided equally between all group members.

This report is directed to people interested in aspect-oriented programming and/or the .NET framework. The report contains information about aspect-oriented programming and how our program uses and manipulates the .NET Intermediate Language of an existing assembly to apply aspects.

Bjørn D. Rasmussen

Casper S. Jensen

Jimmy Nielsen

Lasse Jensen

Acknowledgements

We would like to thank our supervisor Associate Professor Bent Thomsen for many insightful conversations during the work of this thesis and also for helpful comments on the text.

We would also like to thank Microsoft for making available free software at MSDN academic alliance. This made it possible to use the latest .NET development tools during the development of Aspect.NET. The software mostly used was Visual Studio .NET Pro 2003 and Visual SourceSafe 6.0. We would also like to thank Microsoft for all the free information and tutorials about the .NET framework which is accessible from the web. This helped us gather the majority of the information needed for the .NET framework.

Thank you!

Group E4-114

Bjørn D. Rasmussen
Casper S. Jensen
Jimmy Nielsen
Lasse Jensen

Department of Computer Science
Aalborg University
June 11^{th} 2004

CD

This chapter contains information about the CD, which has been attached to the report. The attached CD is supposed to work as documentation for the development of Aspect.NET. Figure 1 illustrates the content of the CD. Please note that not all subdirectories are listed.

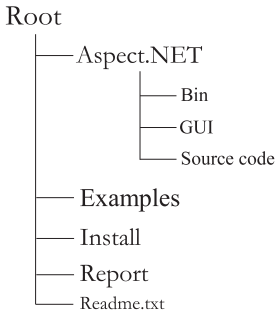


Figure 1: Content of CD.

For optimal use of the attached CD, please read the Readme.txt file, which is located at the root of the CD.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A journey of a thousand miles, must begin with a single step

— Lao-Tsu

Every skilled programmer knows that code repetition should be avoided. Still there exists behavior which is simply not possible to express without repeating code in today's widely used programming languages. Consider that we e.g. would like to perform some action every time an object's state changes. In an object-oriented programming language this would be done by calling a separate method which notifies a listener every time a method is called that affects data inside the object (e.g. Set-methods in Java or C#). This scatters the same code in all state-changing methods of a class while at the same time introduces behavior that is not related to the class and thereby complicating later refinement.

Code repetition (or code scattering) is a result of traditional languages' inability to modularize all concerns that exist in software development, and programmers are therefore forced to implement multiple concerns in a single module. A module in this case should, however, not only be thought of as a class in object-oriented programming, but instead as a wide range of languages' ways of providing encapsulation which can either be classes, procedures, functions, methods etc. Often the concerns that cannot be modularized to a single module are concerns which cut across several modules like e.g. logging, security, synchronization, tracing etc. Given that some concerns may be scattered across several modules makes later refinement very time-consuming and error-prone.

Aspect-oriented programming (AOP) has been designed for what is called "separation of concerns" such that behavior that is not related to the design or implementation of a module is being kept separate. This way the concerns (or behavior) that would normally be scattered in a module or across several modules will also be modularized. AOP applies to all programming languages which have a key abstraction and composition mechanism that is rooted in some form of generalized-procedure [14]. The number of languages which fall into this category is quite significant.

The dominant AOP technology at the moment is AspectJ[1] which allows aspects to be applied to Java[2] applications. AspectJ allows programmers to define so-called pointcuts which capture concerns that cut across a system's modularity. Each of these pointcuts has an advice which is an implementation of a cross-cutting concern. AspectJ thereby modularize code that normally would be scattered across several classes.

1.1 Motivation

AspectJ is a great tool for applying aspects to Java applications but the fact that it is restricted to the Java language is quite a limitation. Many other languages are used in software development today and being able to apply aspects to any of these languages would give a wider range of developers the ability to apply aspects to their software components. One way of allowing developers the ability to apply aspects is by creating an AspectJ-like tool for each of the programming languages which are available, but this would be a cumbersome approach. Many of the traditional and also recent languages are, however, being made available on multi-language platforms such as Java's virtual machine (JVM) [35] or the common language runtime (CLR) [20] for Microsoft's .NET framework[3]. These multi-language platforms make use of high-level intermediate assembly languages which have been designed to incorporate a broad range of language constructs that are used in various programming languages. Developing language compilers for these multi-language platforms is thereby simplified, since the intermediate language has native support for many of the language constructs which one may require.

The intermediate language used by multi-language platforms is relatively high-leveled and possible to reverse-engineer. A program can thereby be manipulated after it has been compiled and this makes it possible to weave aspects into existing programs at the intermediate language level. Since all languages on these platforms are compiled to a common intermediate language, it is then possible to weave aspects into any language that compiles for these platforms. The goal of this project is therefore to design an AspectJ-like language and compiler which supports aspect weaving in a wide range of languages on a multi-language platform, such that developers can benefit from AOP regardless of their high-level language. Given that aspects are woven regardless of high-level language also makes it possible to mix languages, such that some concerns can be implemented in one language and other concerns in another language. One can then choose to implement some algorithms of an application in a functional language, while the rest of the application remains implemented in an object-oriented language. The language independent aspect weaver can then be used to replace calls to algorithms written in the object-oriented language with calls to algorithms written in the functional language and vice versa

Although the JVM is said to be a multi-language platform, it has several restrictions that prevents it from becoming a true multi-language platform [20] and therefore the .NET platform is the targeted platform in this project. The

[1] http://www.aspectj.org
[2] http://java.sun.com
[3] http://msdn.microsoft.com/netframework

aspect-oriented language and compiler being developed in this project is hence called Aspect.NET.

1.2 Prior work

The need for proper separation of concerns has been pointed out in several articles [19], [6], [1] and has resulted in numerous AOP technologies. AOP was first introduced by Kiczales et al. [14], who found that all languages which include some form of generalized-procedure (thereby procedural-, object-oriented- and functional languages) do not provide users with proper separation of concerns. One of the technologies that has emerged from this observation is AspectJ [15] which is designed as an extension to the Java language. Because of AspectJ's wide use, other AOP-technologies often compare the set of features they provide by comparing them to AspectJ.

On the .NET platform, efforts have been made to also support aspect weaving in .NET languages. AspectC# [16] is one of the projects which have been started to support aspect weaving in the C# language and supports a wide range of the features that are available in AspectJ. Unlike AspectJ, AspectC# does not extend the C# language but instead works as a preprocessor which weaves the aspects into the C# source files before the C# compiler is called. AspectC# uses an XML-document to define the calls (or pointcuts) which should be intercepted. As its name states, AspectC# is restricted to one language but attempts have also been made to support cross language aspect weaving for .NET. Among those are Weave.NET [17] which lets users define aspects that intercept calls in .NET's common intermediate language (CIL). Since Weave.NET operates on CIL it thereby supports all .NET languages. This approach, however, has the affect that users not only need to know their high-level language but also CIL. In addition, Weave.NET only supports a very limited number of the features which are available in AspectJ and the capture of crosscutting concerns is therefore quite restricted. Efforts have also been made to support cross-language aspect weaving in so-called dynamic aspect weavers on the .NET platform but this is at the cost of loss of platform independence.

1.3 Thesis outline

Part 1 of this thesis starts by introducing the technologies that are relevant for the design of Aspect.NET. The first chapter of part 1 introduces the .NET framework and in particular how it archives language independence and enables language interoperability. The second chapter outlines how aspect-oriented programming assists developers in capturing and modularizing crosscutting concerns. The last chapter of part 1 introduces other aspect-oriented technologies, which have served as inspiration for the design of Aspect.NET.

Part 2 outlines the design of the Aspect.NET language and compiler. The first chapter of part 2 investigates different approaches for allowing cross-language aspect weaving and concludes which approach will be used for the design of Aspect.NET. The following chapter describes the syntax and semantics of the

Aspect.NET language and how it can be used to capture crosscutting concerns.

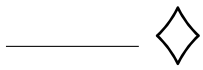
Part 3 consists of two chapters which describes the design and implementation of the Aspect.NET compiler. First an abstract view of the compilers design is outlined and then a description of the weaving process. The implementation of each of the language constructs available in Aspect.NET are outlined as well.

Part 4 describes how to use the Aspect.NET language and compiler. The chapter in part 4 describes how to use the compiler and lists the command line switches that are accepted by the compiler. The chapter also gives concrete examples of how users can use the Aspect.NET compiler to capture and modularize crosscutting concerns. An example of how aspects combined with mixed languages can help to improve application performance and reliability is also supplied.

Part 5 summarizes the results that have been achieved during the design of Aspect.NET and provides ideas for future work that will help to improve Aspect.NET in capturing and modularization of crosscutting concerns regardless of language.

Part I

Prerequisites



This part gives an introduction to the topics that are relevant for the development of Aspect.NET. This includes a chapter describing the .NET framework, and a chapter on aspect-oriented programming. Last, a subset of the existing aspect-oriented technologies are introduced.

Chapter 2

The .NET framework

The beginning is the half of every action

— Greek Proverb

In July 2000 Microsoft introduced the .NET framework as a new development platform for simplifying development of the Windows operating system. The .NET framework encapsulates the numerous APIs that are available in Windows, like e.g. MFC [28], ATL [23], COM [24], etc. thereby allowing uniform access to Windows' resources. The most innovative feature of the .NET framework is, however, the common language runtime (CLR), which is a virtual execution system (VES) where all .NET applications are executed. As its name states, the CLR is a runtime environment which incorporates all languages into a single runtime. The CLR thereby eliminates the language barrier. This is quite an improvement over having to use e.g. Dynamic Link Libraries (DLLs) or interprocess communication like COM to do language interoperability. The .NET framework is also set to eliminate problem with types which has often posed problems when performing language interoperability, especially for floating point values and string encodings. In order to overcome this problem the .NET framework makes use of a common type system (CTS) which is shared among all .NET languages.

Applications for the .NET framework are stored in a so-called managed executable module which is an extension of the Microsoft Windows Portable Executable and Common Object File Format (PE/COFF) [31], [18]. A .NET executable module contains metadata which describe the content of the module like e.g. new types that are being introduced by the module and whether the module has dependencies to other modules. The executable code of a .NET module is based on the instruction set of the CLR which is the Common Intermediate Language (CIL), also called Microsoft Intermediate Language (MSIL). Before a .NET executable module is executed, it is compiled into native code which is often referred to as Just-in-time (JIT) compiling. Figure 2.1 illustrates the compilation process and execution of .NET applications.

The .NET framework is an instance of the Common Language Infrastructure (CLI) standard which is outlined in the following section.

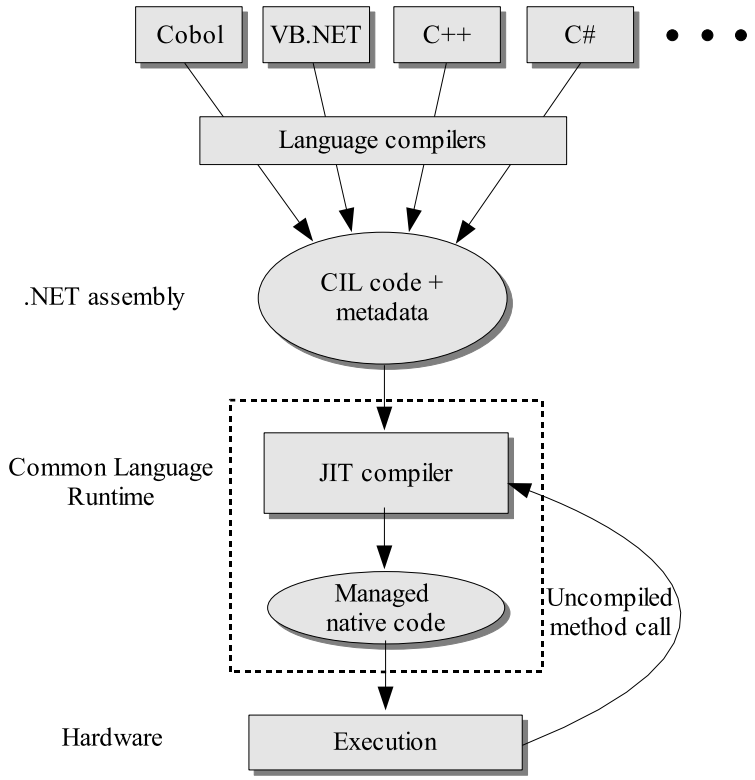


Figure 2.1: Overview of the compilation and execution of .NET applications.

2.1 The Common Language Infrastructure

The .NET framework is built according to the common language infrastructure which is a specification for executable code and the runtime system that executes the code. The CLI specification [8], [9], [10], [11], [12] has been accepted as an international standard by the International Organization for Standardization[1] (ISO) and the European Computer Manufacturer's Association[2] (ECMA) and consists of the following parts:

- **Partition I: Concepts and Architecture**
  This part of the standard describes the architecture of the CLI along with a description of the CTS, VES and the common language specification (CLS).
- **Partition II: Metadata Definition and Semantics**
  A description of the structure and semantics of metadata including its file layout and content.

[1] http://www.iso.org/
[2] http://www.ecma-international.org

- **Partition III: CIL Instruction Set**
  A detailed description of the instruction set used by the VES.
- **Partition IV: Profiles and Libraries**
  The CLI includes a common library that can be used by CLI languages. The classes, value types and interfaces of the CLI are outlined in this part.
- **Partition V: Annexes**
  This part has examples of applications written in CIL as well as a description of the tools that are available to manipulate CIL. Guidelines for implementing cross-language libraries are also provided.

The core components for allowing language integration in the CLI standard are the CLS, the CTS, VES, CIL and metadata which are elaborated in the following five sections.

2.2 The Common Language Specification

One of the goals of the CLI is to support language integration, such that e.g. types defined in one language can be used in other languages. The CLS defines the rules that make this a reality. There exist three different views of CLS compliance[3]: Framework, Consumer and Extender. These views are described in further detail below:

- **Framework**
  A framework is a library consisting of CLS compliant code. These frameworks are intended for use by a wide range of languages and tools including CLS Consumer and Extender. The following CLS guidelines should be followed:
  - Names commonly used as keywords in programming languages should be avoided.
  - It cannot be expected that users are able to author nested types.
  - It can be assumed that implementations of methods with identical name and signature on different interfaces are independent.
  - It cannot be assumed that value types are initialized based on specified initialized values.
- **Consumer**
  A Consumer is a compiler that generates code for CLI and is designed to use such libraries, but does not produce or extend library code. Such compilers are referred to as "Consumers". A Consumer is designed to allow access to all features of the CLS-compliant framework. CLS Consumer tools are expected to:
  - Be able to call any method or delegate which are CLS-compliant.
  - Have a mechanism for calling methods which have names that are keywords in the programming language.

[3]For further information on these views and CLS rules, see [8].

  - Be able to call distinct methods, supported by a type, which are identical in the form of having the same name and signature.
  - Allow access to nested types.
  - Any CLS compliant fields should be both readable and writable.
  - Create any CLS-compliant type.
  - The get and set methods of CLS compliant properties should be accessible.

  It is not a requirement that any CLS Consumer tool deliver support for the following:

  - It is not necessary that a CLS Consumer tool supports the creation of new types or interfaces.
  - Furthermore it is not necessary that a CLS Consumer tool is able to initialize metadata on fields or parameters, besides static literal fields.

- **Extender**
  An Extender is a compiler that generates code for the CLI and is designed to use, produce and extend such libraries. Such compilers are referred to as "Extenders". This means that besides the requirements which exist for the CLS Consumer, a CLS Extender must additionally follow these rules:

  - A CLS Extender must be able to define new types which extends CLS compliant base classes.
  - A CLS Extender must be able to provide independent implementations for methods of all interfaces which is supported by a type.
  - Defining types with names that are keywords in the language.
  - An Extender must be able to implement CLS compliant interfaces.
  - Attributes must be able to be placed on appropriate metadata elements.

  It is not a requirement that a CLS Extender supports the following rules:

  - A CLS Extender need not have support for definition of CLS compliant interfaces.
  - Furthermore a CLS Extender does not have to support for definition of nested types.

Note that the CLS rules apply only to those items which are visible outside the defining assembly. This means that the rules apply to:

- Public classes.
- Public members of public classes and members accessible to derived classes.
- Parameter and return types of public classes and methods accessible to derived classes.

2.3 The Common Type System

The CTS is one of the core components for ensuring language interoperability because it describes how data should be interpreted. Types in CTS are either reference types or value types. Value types represent simple bit patterns for what is often referred to as primitive types in programming languages like C [5] and can e.g. be integers and floats. Reference types, on the other hand, are self-typing because they describe their own representation and their identity distinguishes them from other reference types. Value types are types that are allocated on the stack whereas reference types are heap allocated. Figure 2.2 illustrates a tree which contains all the types existing in the CTS. In this tree the different subtypes of value types and reference types can be seen.

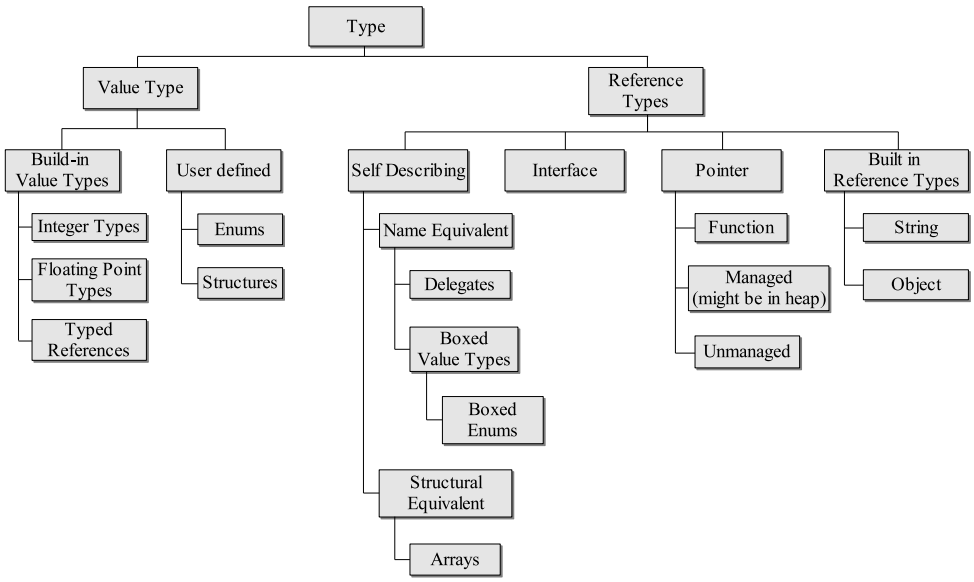


Figure 2.2: CTS type tree.

Value types can also be used as reference types, by the use of boxing. This means that e.g. an integer can be initialized as an object and thereby be used in operations that would normally be restricted to reference types. Unboxing is then used to transfer a reference type to a value type if the reference type supports such an operation. The CTS provides a "boxing parent" for all value types which is shown in Table 2.1.

2.4 The Virtual Execution System

Executable code in a PE file uses the CIL instruction set (see section 2.5) which is executed by the CLI's Virtual Execution System (VES). VES is a stack-based computing machine, meaning that values used in execution are pushed onto and

Built-in types			
Name in CIL	**CLS type**	**Name in Class library**	**Description**
bool	yes	System.Boolean	True/False value
char	yes	System.Char	Unicode 16-bit char
object	yes	System.Object	Object or boxed type
string	yes	System.String	Unicode string
float32	yes	System.Single	IEC 60559:1989 32-bit float
float64	yes	System.Double	IEC 60559:1989 64-bit float
int8	no	System.SByte	Signed 8-bit integer
int16	yes	System.Int16	Signed 16-bit integer
int32	yes	System.Int32	Signed 32-bit integer
int64	yes	System.Int64	Signed 64-bit integer
native int	yes	System.IntPtr	Signed integer, native size
native unsigned int	no	System.UIntPtr	Unsigned integer, native size
typedref	no	System.Typed-Reference	Pointer plus runtime type
unsigned int8	yes	System.Byte	Unsigned 8-bit integer
unsigned int16	no	System.UInt16	Unsigned 16-bit integer
unsigned int32	no	System.UInt32	Unsigned 32-bit integer
unsigned int64	no	System.UInt64	Unsigned 64-bit integer

Table 2.1: Built-in types.

popped from an evaluation stack. Code executed by the VES is referred to as managed code because the VES provides services like garbage collection of heap allocated instances, array index checking and exception handling.

The VES supports a limited number of data types which are described in Table 2.2. The data types described as native size use the default size on the hardware architecture where they are executed. On a IA-32 processor, like a Pentium, this size would be 32-bit whereas an IA-64 processor would use 64-bit. The native unsigned int data type can also be used to represent unmanaged pointers if users wish to explicitly delete heap allocated instances and thereby not make use of the VES's garbage collecting service. The VES's `O` data type is used to reference "the beginning" of objects or arrays while the `&` data type can be used to reference the interior of an object or array. Since objects and arrays are garbage collected and may be moved means that the value of `O` and `&` data types may change during execution.

When the VES loads a PE file for execution it scans each method for the following information:

- Instructions to execute and exception handlers related to those instructions.

Data type	**Description**
int8	8-bit 2's complement signed value
unsigned int8	8-bit unsigned value
int16	16-bit 2's complement signed value
unsigned int16	16-bit unsigned value
int32	32-bit 2's complement signed value
unsigned int32	32-bit unsigned value
int64	64-bit 2's complement signed value
unsigned int64	64-bit unsigned value
float32	32-bit IEC 60559:1989 floating point value
float64	64-bit IEC 60559:1989 floating point value
naive int	native size 2's complement signed value
native unsigned int	native size unsigned value
F	native size floating point number
O	native size object reference to managed memory
&	native size managed pointer

Table 2.2: Data types supported by the VES.

- Method signature specifying the methods return type, number of arguments and the type of those arguments.
- The maximum size of the evaluation stack.
- The list of local variables, their type and whether local variables should be initialized to null.

Once the VES has loaded the above information it is ready to execute the PE file. Figure 2.3 illustrates the VES state model, which includes threads of control, method states and multiple heaps in a shared address space. For each method invocation (or created stack frame) the VES needs to store the current method state such that it later on can resume execution. The current method state consists of the following:

- The instruction pointer's (IP) current position.
- The current evaluation stack.
- An array of local variables in order to preserve their values.
- An array of arguments.
- A so-called method information handler containing a method signature and the types of local variables.
- A local memory pool for dynamically allocated objects.
- A return state handle for restoring the method's state. In compiler terms, this is referred to as a dynamic link.
- A security descriptor to record security overrides.

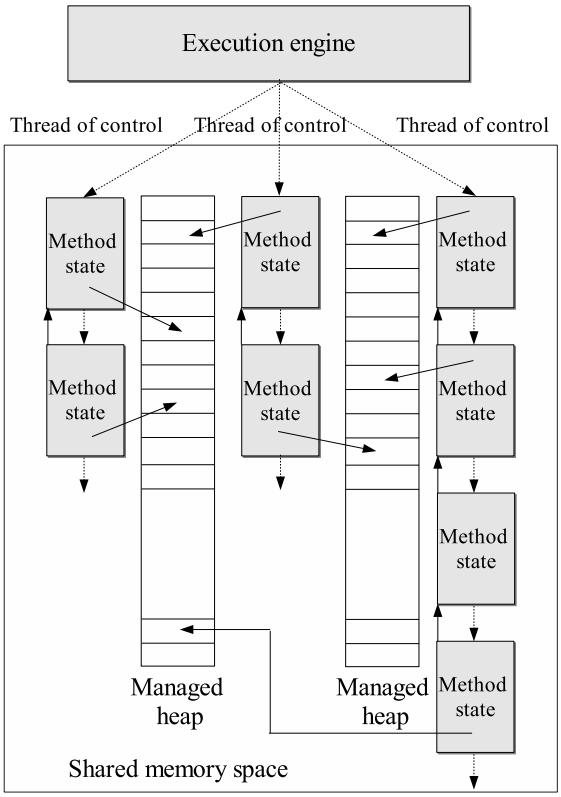


Figure 2.3: Virtual Execution System state model.

2.5 The Common Intermediate Language

All CLI compliant compilers translate their high-level language to the common intermediate language (CIL) which is the assembly language of the VES. The CIL is, however, a far more high-level assembly language than what developers are used to because it not only describes machine instructions; but also constructs in programming languages. One can therefore also choose to see CIL as a meta-language since it describes other languages. Below is a simple "Hello World" example written in the CIL assembly language:

```
.assembly extern mscorlib {}
.assembly hello {}

.method static public void main() cil managed
{ .entrypoint
  .maxstack 1
  ldstr "Hello World"
  call void [mscorlib]System.Console::WriteLine(class System.String)
  ret
}
```

The above example defines an assembly called `hello` which has an external reference to `mscorlib` which is the class library that comes with the .NET framework. A method called `main` is defined which is the entrypoint of the assembly denoted by `.entrypoint`. The maximum size of the stack when the method is called, is then defined and in this case set to 1. Then comes the CIL instructions which are the actual executable code in the assembly. The string "Hello World" is pushed onto the stack and a static method is called that prints the string to the console.

2.5.1 Language constructs in CIL

Since the goal of CIL is to describe all languages, it incorporates a wide range of language constructs that exists in other programming languages. Partition II [9] of the CLI standard gives a detailed description of all these language constructs. The design of CIL is to a great extend based on supporting the object-oriented paradigm. CIL supports classes which must inherit from exactly one class, except the built-in class `System.Object`. The types of methods supported for classes include both static, virtual and instance methods, but global methods that are not related to a class are also supported by CIL. Every class is also able to implement zero or more interfaces.

2.5.2 The CIL instruction set

The CIL also has an instruction set which is described in Partition III [10] of the CLI standard. The CIL is similar to the instruction set seen in e.g. x86 processors although some of the instructions in the CIL are a bit more abstract. One example is that the CIL instruction set has the notion of `object` which is

unfamiliar to the x86 instruction set. The CIL currently consists of about 220 instructions and the following four sections give a brief overview of the most fundamental instructions.

Load and store instructions

Load instructions push values from memory onto the evaluation stack while store instructions pop values from the evaluation stack and store the values in memory. Here is a list of some of the basic load and store instructions:

- `ldc.i4 v` - Pushes a 4-byte integer constant with the value `v` onto the stack.
- `ldnull` - Pushes a null reference onto the stack.
- `ldloc n` - Pushes the `n`-th local variable onto the stack.
- `ldloca n` - Pushes the address of the `n`-th local variable onto the stack.
- `ldind.i4` - Dereferences the top element on the stack and pops. Then pushes the address' value onto the stack.
- `ldarg n` - Loads the `n`-th argument onto the stack.
- `stloc n` - Stores the stack's top element in the `n`-th local variable and pops the stack.
- `stloca n` - Stores the address of the stacks top element in the `n`-th local variable and pops the stack.
- `stind.i4` - Stores the topmost value on the address and pops.
- `starg n` - Stores the value on top of the stack in the `n`-th argument and pops the stack.

The following example, which swaps the arguments `a` and `b`, illustrates how to use the load and store instructions[4]:

```
.method public hidebysig static void Swap(int32& a,
                                          int32& b) cil managed
{
  // Code size       11 (0xb)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  ldarg.0
  IL_0001:  ldind.i4
  IL_0002:  stloc.0
  IL_0003:  ldarg.0
  IL_0004:  ldarg.1
  IL_0005:  ldind.i4
  IL_0006:  stind.i4
```

[4]Note that the example uses call by reference.

```
  IL_0007:  ldarg.1
  IL_0008:  ldloc.0
  IL_0009:  stind.i4
  IL_000a:  ret
} // end of method Test::Swap
```

Arithmetic instructions

Arithmetic instructions in CIL pop the top two elements on the stack and perform its calculation. CIL supports the following arithmetic instructions:

- `add` - Adds the two top elements and leaves the result on top of the stack.
- `sub` - Subtracts the stack's second top-most element from the top element and leaves the result on top of the stack.
- `mul` - Multiplies the two top elements and leaves the result on top of the stack.
- `div` - Divides the stack's second top-most element with the top element.
- `div.un` - Same as `div` but for unsigned integers only.
- `rem` - Leaves the remainder on top of the stack after having performed a modulo operation.
- `rem.un` - Same as `rem` but for unsigned integers only.

In addition to the above arithmetic instructions, CIL has the instruction `neg` which pops the top element of the stack and performs an negation of the sign and pushes the new value onto to stack. Most arithmetic instructions are typeless, meaning that they expect both of the stack's top elements to be of the same type when executed. The `div.un` and `rem.un` are, however, exceptions.

The following example illustrates how two local integer variables are subtracted and stored in another local variable.

```
.method public hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size       9 (0x9)
  .maxstack  2
  .locals init (int32 V_0, int32 V_1, int32 V_2)
  IL_0000:  ldc.i4.1
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.2
  IL_0003:  stloc.1
  IL_0004:  ldloc.0
  IL_0005:  ldloc.1
  IL_0006:  sub
  IL_0007:  stloc.2
  IL_0008:  ret
} // end of method Test::Main
```

Control flow instructions

The CIL supports three types of branching instructions: Unconditional branching, conditional branching and comparative branching instructions. Every type of branching instruction is available in two forms, a branching instruction that can jump to an offset within an 8-bit range, or an offset available within a 32-bit range denoted by <`length`>. E.g. the `beq` comparative instruction is available as `beq` for int32 and `beq.s` for int8. The instructions for the three types of branching instructions are listed here:

- Unconditional branch instructions
  Leave the stack unchanged and jump to a new instruction. These are e.g. used for the break statement in C#.
  - `br.`<length> *target* - Jumps to offset *target*.
- Conditional branch instructions
  Pop the stack and jump to a new instruction based on the popped value.
  - `brfalse.`<length> *target* - Pops the stack and jumps to *target* if value is 0, null or false.
  - `brtrue.`<length> *target* - Pops the stack and jumps to *target* if value is non 0, non-null or non-false.
- Comparative branching instructions
  Pop the two top stack elements and jump to a new instruction if the comparison evaluates to true. The value first loaded onto the stack is the left-hand side and last value loaded onto the stack is the right-hand side.
  - `beg.`<length> - Branch on equal.
  - `bne.`<length> - Branch on not equal.
  - `bne.un.`<length> - Branch on not equal where both values are interpreted as unsigned.
  - `bge.`<length> - Branch on greater than - equal.
  - `bge.un.`<length> - Branch on greater than - equal where both values are interpreted as unsigned.
  - `bgt.`<length> - Branch on greater than.
  - `bgt.un.`<length> - Branch on greater than where both values are interpreted as unsigned.
  - `ble.`<length> - Branch on less than - equal.
  - `ble.un.`<length> - Branch on less than - equal where both values are interpreted as unsigned.
  - `blt.`<length> - Branch on less than.
  - `blt.un.`<length> - Branch on less than - equal where both values are interpreted as unsigned.

Just as for arithmetic instructions, many of the comparative branch instructions are type-less thereby assuming that both values used in the comparison are already of the same type.

Another control flow instruction is the `switch` instruction which uses a jump table for finding the correct branching offset. The following example illustrates a `switch` operation on the local variable indexed 0 that is assigned the value 2. The `switch` instruction's jump table accepts the values 1 and 2:

```
.method public hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  2
  .locals init (int32 V_0, int32 V_1)
  IL_0000:  ldc.i4.2
  IL_0001:  stloc.0
  IL_0002:  ldloc.0
  IL_0003:  stloc.1
  IL_0004:  ldloc.1
  IL_0005:  ldc.i4.1
  IL_0006:  sub
  IL_0007:  switch    ( IL_0016, IL_0022)
  IL_0014:  br.s       IL_002e
  IL_0016:  ldstr      "1"
  IL_001b:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0020:  br.s       IL_002e
  IL_0022:  ldstr      "2"
  IL_0027:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_002c:  br.s       IL_002e
  IL_002e:  ret
} // end of method Test::Main
```

Method invocation instructions

Since CLI supports several languages which have different types of methods, the CIL has multiple instructions for invoking methods.

- `call` *`method`* - Calls either a static method, instance method, virtual method or global function based on a metadata token indicated by *`method`*. For virtual and instance methods the `call` instruction can be followed by `instance` which indicates a static method lookup (also called an early bound call). When `instance` is used, a "this" pointer is expected on top of the stack. If a method has parameters, these are loaded onto the stack after the "this" pointer in a left to right order. `call` can also be preceeded by `tail.` meaning that the caller's stack frame should be popped before transferring control.
- `callvirt` *`method`* - This instruction is quite similar to the `call` instruction although `callvirt` always requires a "this" pointer and the method lookup is performed at runtime instead of compile time. `callvirt` can also be preceeded by `tail.`

- `calli` *`entryPoint`* - This instruction expects a pointer to a native code method entry point on top of the stack and if the method requires parameters, these are loaded onto the stack before the function pointer. A "this" pointer can also be loaded on the stack before the parameters of the native method. Just as the other call instructions, the `calli` also supports the `tail.` prefix.

2.6 Metadata

Section 2.3 gives a brief overview of the Common Type System. However, sharing types between compilers is enabled by metadata. It is important to clarify that metadata is not a new concept introduced by the Microsoft .NET framework - metadata have existed for several years. Before the introduction of the .NET framework, metadata often existed in language specific files and were written by the developers in e.g. an Interface Definition Language (IDL) as used by e.g. COM and CORBA. Using metadata this way has over the years caused several problems, below is listed a few:

- Due to the fact that the metadata have been written in language specific files, sharing this information across language boundaries has been rather difficult.
- When developers write metadata, it is often stored in auxiliary files, resulting in inconsistencies and versioning problems.
- Often the IDLs only allow the developer to specify the syntax of an interface, but not its semantics.

The metadata system introduced in the .NET framework, tries to solve all of the above problems. Based on previous metadata systems, some of the main concerns with the .NET metadata systems have been to ensure that metadata allows types defined in one language to be used in another language. Metadata together with CTS are the technologies that ensures this language interoperability. The execution engine needs information about e.g. memory management and security. Therefore metadata additionally stores this information.

Metadata is auto-generated and stored side-by-side together with the CIL code in the PE file. This eliminates the problems with versioning and inconsistency problems. Also the developer does not have to learn an additional language, in the .form of e.g. IDL, in order to gain the advantages that exist by having a metadata system.

The metadata contains information about assemblies, types and attributes. With assemblies information like name, version and types that are exported and dependencies is stored. With regard to types, information like name, visibility and members is stored. Attributes are either those predefined in the .NET framework, or the custom attributes created by the developer. Attributes allow the developer to additionally describe the specific program elements.

Chapter 3

Aspect-Oriented Programming

Everything should be made as simple as possible, but not simpler.
— Albert Einstein

Traditional programming languages provide ways to encapsulate concerns into either functions or modules which results in reduced complexity in writing and maintaining complex applications. Some concerns, however, do not only relate to a single function or module and are therefore scattered across several functions or modules. The following sections outline the basic ideas behind AOP and how it will supplement traditional programming in properly separating concerns.

3.1 Concerns

A software system can be viewed as a combined implementation of multiple concerns. A typical system may consist of several kinds of concerns including business logic, data persistence, logging, debugging, authentication, security and many more.

In the simplest form, there are core concerns, which are components that provide the actual functionality of the software. Additionally, there are system-level concerns which can be e.g. logging, debugging and authentication that tend to affect several other concerns. For instance, a logging feature implemented into a program is likely to be implemented in several classes. Each class will have to have its own code for logging, making the classes less specialized, resulting in making it very hard to predict what effect changes to the logging code will have. This phenomenon is called crosscutting concerns. Normally when referring to crosscutting, two types of crosscutting exist:

- Static crosscutting

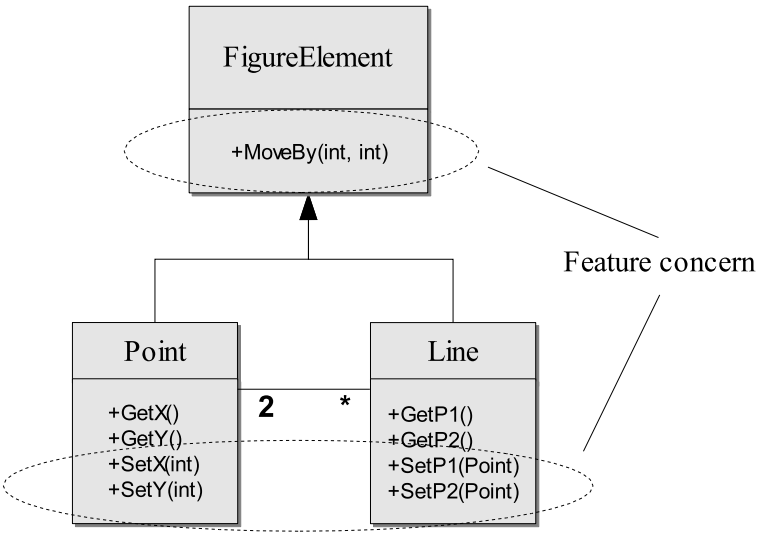


Figure 3.1: A crosscutting concern.

  - Static crosscutting makes it possible to define new operations on existing types and is referred to as static crosscutting because it affects the static type signature of the program.

- Dynamic crosscutting
  - Dynamic crosscutting makes it possible to define new operations to run at certain well defined points in the execution of the program.

3.1.1 Crosscutting concerns

Two concerns are crosscutting, if the methods related to these concerns intersect and cannot be separated from each other. On Figure 3.1 a crosscutting concern is illustrated.

The UML diagram seen on Figure 3.1 illustrates a simple figure editor which has two concerns:

- Data concerns
- Feature concerns

Keeping track of each FigureElement (data concern) and updating the display whenever a FigureElement has moved (feature concern). The data concern is encapsulated due to the object-oriented (OO) design. The feature concern, however, must appear in every movement method, thereby crosscutting the data concern.

The OO-paradigm only allows to encapsulate one concern, and other concerns cannot be encapsulated within the dominant modules. Resulting in scattering the concerns cross many modules and tangle with one another.

To illustrate this using an example of a logging concern, the concern needs to intertwine with other concerns in order to be able to perform its operation. The logging concern can e.g. consist of a `Logger` class which implements the interface:

```
public interface Logging{
  public static void logEntry(string func);
  public static void logExit(string func);
}
```

Here the `Logger` implements two methods `logEntry(String func)` and `logExit(String func)` which respectively logs the entry and the exit of a method.

In order to use the `Logger` class each class that should be logged, needs to know about the `Logger` class. An example of usage of the `Logger` class is:

```
public class Main
{
  public void foo()
  {
    Logger.logEntry("foo()");
    Console.WriteLine("Hello world!");
    Logger.logExit("foo()");
  }

  public double bar(double x, double y)
  {
    Logger.logEntry("bar(double, double)");
    return x + y;
    Logger.logExit("bar(double, double)");
  }
}
```

Here the logging methods are called every time a method is called and when it exits. However, the logging methods need to be included and written into all the methods by the programmer of the method.

Another problem arises in the `Logger` class. In the above example where there is an error in the `bar()` method. Here the `logExit()` is never reached because the programmer put the method after the return statement. This error may not be caught until the log-file is consulted.

When this error needs to be corrected the programmer has to change the code in every method where the logging methods are called, making error correction harder.

Taking a project of 30 classes with each class having at least 5 methods, there will be over 300 places where the programmer needs to correct the errors.

Figure 3.2 illustrates a code scattering problem in a subset of the classes in a Tomcat Webserver[1]. Each column in the bar chart illustrates a class, where the size of the column illustrates lines of code. The colored lines in each column illustrate the lines of code where logging code needs to be inserted in a normal object-oriented design pattern. This also means that inserting logging code across the entire system can be error prone. This also emphasizes AOP, since in this paradigm it is possible to isolate the logging code from the original system code. This means that the system design becomes more modularized and thereby easier to maintain.

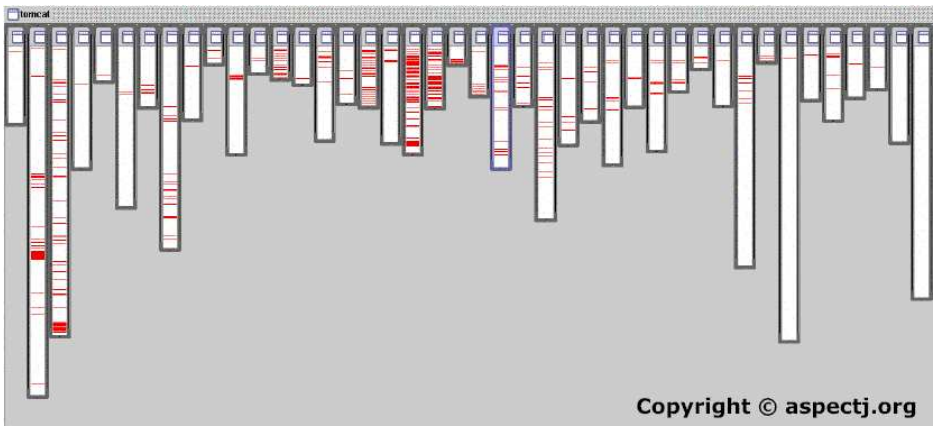


Figure 3.2: Logging example illustrating code scattering.

The code tangling and code scattering affect the software design and development in many ways:

- Poor trace-ability
  - Simultaneously implementing several concerns obscures the correspondence between a concern and its implementation, resulting in a poor mapping between the two.
- Reduced productivity
  - The implementation of several concerns takes the developers off the main concern, leading to a reduced productivity.
- Less code reuse
  - Another system requiring the same functionality from a system implementing several concerns may not be able to readily reuse the module.
- Poor code quality

[1]Note that the figure has been taken from http://www.AspectJ.org

  - Implementing several concerns which results in code tangling, produces code with hidden problems. Furthermore targeting many concerns at once results in one or more concerns that will not receive enough attention.

- More difficult evolution
  - A system implementing several concerns makes it harder to evolve a system over time, since the implementation is not modularized. Modifying a module can lead to inconsistencies and require considerable testing effort to ensure that the changes have not caused bugs.

3.2 Modularizing crosscutting concerns

A solution to crosscutting concerns is AOP, which lets users implement individual concerns and combines these implementations to form the final system.

AOP involves three distinct development steps [30]:

- Aspectual decomposition
  - Identify the crosscutting concerns and separate them from the common concerns.
- Concern implementation
  - Each crosscutting concern is to be implemented separately.
- Aspectual recomposition
  - The recomposition process, also called *weaving*, composes the final system by weaving the aspects into the code.

These three steps will be explained in the following sections.

3.2.1 Aspectual decomposition and concern implementation

The aspectual decomposition is the first process of identifying each concern and deciding which ones are crosscutting. Logging was introduced in the previous section as a concern which is crosscutting, since it cuts across several modules in the system. However, in terms of programming it requires much more than just identifying the different aspects of concern. It additionally requires an ability to express those aspects of concerns in a precise manner.

When each concern is identified, it is implemented separately in an aspect. This means that the crosscuts identified in the logging example are implemented separately.

3.2.2 Aspectual recomposition

In the aspectual recomposition, the weaver plays an important role in an aspect-oriented programming paradigm. Its job is to "weave" the target implementation and aspects together into an executable. The weaver parses the aspect program and collects information about the crosscutting concerns referenced by the program. After this, the weaver locates the referenced points in the code. Now the aspect weaver is ready to weave the implementation of crosscutting concerns into the existing code. On Figure 3.3 the weaving process is illustrated.

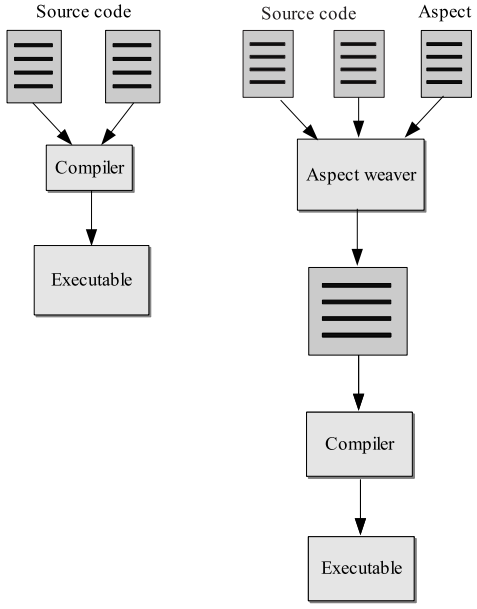


Figure 3.3: Normal compilation and AOP compilation model.

The two most commonly used aspect weaving techniques are static aspect weaving and dynamic aspect weaving, which are outlined in the following two sections.

Static aspect weaving

Static aspect weaving refers to modifying the source code at compile time by in-lining the aspect code into the target source. This means that the additional abstraction level introduced by aspect-oriented programming does not have a negative impact on the target programs performance. Examples of static weavers are AspectJ and Weave.NET, which are described in sections 4.1 and 4.3.

Dynamic aspect weaving

Another approach to aspect weaving is dynamic aspect weaving, where weaving occurs at runtime. Therefore an important requirement for using dynamic aspect weaving is the explicit existence of aspects at weave and runtime. The fact that aspects are woven at runtime makes it possible to add, remove and replace aspects while an application is running - and without having to restart the system. Dynamic weaving can e.g. be used to assist debugging, since new aspects can be inserted while the application is running. Although dynamic aspect weaving provides a higher degree of flexibility, than what is provided with static aspect weaving, it can impose quite a performance penalty since checks has to be performed to see whether an aspect is active or not. A example of a dynamic aspect weaver is described in section 4.4.

3.3 Traps in aspect-oriented programming

Although AOP closes some of the gabs that traditional languages fail to capture, there are problems related to applying aspects to software components. One of the problems that AOP impose when used in collaboration with traditional languages like e.g. object-oriented programming languages (OOPL) is that it breaks encapsulation which is the primary goal of OOPL. When using AOP the entire behavior and manipulation of an object is no longer within that object but is now also described separately in aspects that apply to that object.

Aspect-oriented programming may also complicate the search for errors in an application because part of the application's source code (the aspects) are woven into the remaining source code during compilation. So if e.g. a particular function in a procedural language is causing an application to crash; then investigating the sequence of statements that are executed in this function may not unveal the error, because the error that is causing the crash is in fact an aspect, that is woven into this function during compilation.

Another problem that already poses a significant problem within the traditional languages, is multi-threading which gets a whole new dimension when introducing aspects in existing software components. New languages like Java and C# provide object-level synchronization but when aspects are introduced parts of the object's description and behavior are written in those aspects and these should also obey an object's rules for synchronization.

Chapter 4

Aspect-Oriented Technologies

Power is nothing without control.

— anonymous

This chapter describes the state-of-the-art technologies within aspect-oriented programming. So far Java is the best supported platform for applying aspects to software components, and it is mainly the AOP language AspectJ[1] which attracts the most attention at the moment.

Even though the Microsoft .NET framework is relatively new, AOP technologies are also starting to emerge for this platform. The three most interesting technologies for the .NET platform are Weave.NET which offers language-independent aspect weaving, and AspectC# which offers static and dynamic crosscutting for the C# programming language. The last one is the Dynamic AOP-Engine for .NET which offers weaving and un-weaving at runtime.

Each of these technologies for the Java and .NET platform are outlined in the following sections.

4.1 AspectJ

AspectJ is the most widely used aspect-oriented programming language. AspectJ weaves aspects into Java applications by using its own Java compiler which includes an aspect-oriented extension for the Java language. AspectJ is therefore a static aspect weaver, since the weaving process occur at compile time. The AspectJ language supports the following language constructs:

- Join points
- Pointcuts

[1] http://www.aspectj.org

- Advices

Each of these are explained respectively below.

4.1.1 Join point

A join point is a well-defined point in the controlflow of a program. Some features of this are listed below:

- Method and constructor call join points
- Method and constructor execution join points
- Field get and set methods join points
- Exception handling join points

For example, a method call join point is a point in the flow where a method is called. The lifetime of a join point includes all the actions that the method comprises of - each method call is one join point.

4.1.2 Pointcut

A pointcut designator (or simply put pointcut) selects particular join points by filtering out a subset of all join points, based on a defined criterion. An example of a pointcut is:

```
call ( public void Point.setX ( int ) )
```

Here the pointcut filters out all the join points except the calls to the public method `setX(int)` in the class `Point`. Pointcuts can be build out of other pointcuts by the following boolean operators:

- && – and
- || – or
- ! – not

This can create pointcuts like this:

```
call ( public void Point.setX ( int ) ) || call ( public void
Point.setY ( int ) )
```

This pointcut filters out all the joint points except the calls to the public method `setX(int)` or `setY(int)` in the class `Point`. With AspectJ a set of primitive pointcut designators is available. A subset of the most important primitive pointcut designators is listed below:

- `args(type or ID, ...)`

  - Every join point where the arguments are instances of `type` or `ID`s type.

- `target(type or ID)`
  - Every join point where the target executing object is an instance of `type` or `ID`s type.
- `call(signature)`
  - Every call to methods or constructors matching `signature` at the call site.
- `reception(signature)`
  - Every reception of methods matches `signature`.
- `execution(signature)`
  - Every execution of methods or constructors matching `signature`.
- `within(typePattern)`
  - Every join point from the code that has been defined in a type in `typePattern`.
- `cflow(pointcut_designator)`
  - Every join point in the code control flow of each join point `P` picked out by `pointcut_designator`, including `P` itself.

For a more in-depth description on each of these pointcut designators, please see the following articles [15] and [3].

Wildcards

AspectJ allow a simple wildcard mechanism. The following syntactical wildcard symbols are available:

- *
- ..

These two symbols can be utilized in the following situations:

- `call(* PointExample.*(..))`
  - Matches calls to any method defined in the class `PointExample`.
- `call(PointExample.new(..))`
  - Matches calls to any constructor for an object of type `PointExample`.
- `call(public * com.aspectJ.communication.*.*(..))`

  - Matches calls to any public method defined in any type in the `com.aspectJ.communication` package.

- `call(* PointExample.get*())`
  - Matches calls to any method defined in `PointExample` for which the the id starts with `get` and which accepts zero arguments.

Wildcards have multiple advantages, but most important of all, wildcards are easy to use and save the programmer from writing long pointcut definitions.

4.1.3 Advice

An `advice` is an implementation of a crosscutting concern and defines the code that should be executed at join points. Pointcuts are used in the definition of an `advice`. There are three kinds of advices which are listed in the following:

- Before advice
  - The code executes when a join point is reached but before the computation proceeds. The before advice is applied by using the `before()` keyword and is used whenever an aspect should be applied before a method is e.g. called.
- After advice
  - The code executes after the computation of a join point has completed, but before the exit of that join point. The after advice is applied by using the `after()` keyword. This advice is used when an aspect should be executed after the call of a method.

    With respect to the after advice, two special cases emerge, namely `after throwing` and `after returning`. `After throwing` is used when an exception is thrown. `After returning` is used when after is returning normally.
- Around advice
  - The code specified in the advice replaces the current code specified in the join point and executes when the join point is reached and conditions specified in the advice are satisfied. The around advice is applied by using the `around()` keyword.

4.1.4 Aspect

An aspect is the construct that encapsulates crosscutting concerns. They are defined by aspect definitions much like a regular class declaration. An aspect declaration may include pointcuts and advices, as well, as other methods. An example of an aspect is illustrated below:

```
aspect MoveTracking{

  pointcut move():
    call(public void Point.setX(int)) ||
    call(public void Point.setY(int));

  before(): move()
  {
    logWrite(``The point is about to be moved'');
  }

  after(): move()
  {
    logWrite(``The point has been moved'');
  }

  public void logWrite(object data)
  {
    System.out.println(data.toString());
  }

}
```

This aspect is used to track whenever a point is moved. The aspect defines one pointcut named `move()` which takes zero arguments. The pointcut consists of two well defined join points, which intercepts all calls to the two methods `setX(int)` and `setY(int)` from the class `Point`.

The before advice defines that whenever `move()` is invoked, the string "The point is about to be moved" should be printed to the console. `move()` is invoked whenever a call anywhere is made to `setX` or `setY` in the class `Point`.

Similar the after advice defines that whenever `move()` is invoked, the string "The point has been moved" should be printed to the console.

This aspect is used to track whenever a point is moved. The aspect defines a pointcut, which intercepts all the calls to the methods `setX(int)` and `setY(int)` in the class `Point`.

The `before()`- and `after()` advice defines that right before/after the `setX` and `setY` methods are called a message, containing information about "the point is about to be moved", is be printed to the console.

4.2 AspectC#

AspectC# [16] was developed as a master thesis and is an ongoing project. The goal of AspectC# is to enable aspect-oriented features to be used with the C# programming language. AspectC# is furthermore build with a basis on AspectJ, due to the popularity of the AspectJ technology. AspectC# additionally supports a rich set of features from AspectJ, e.g. the before, around and after advices. AspectC# defines aspect by means of an XML document,

in AspectC# called an `Aspect Deployment Descriptor`. Figure 4.1 illustrates the overall architecture of AspectC#.

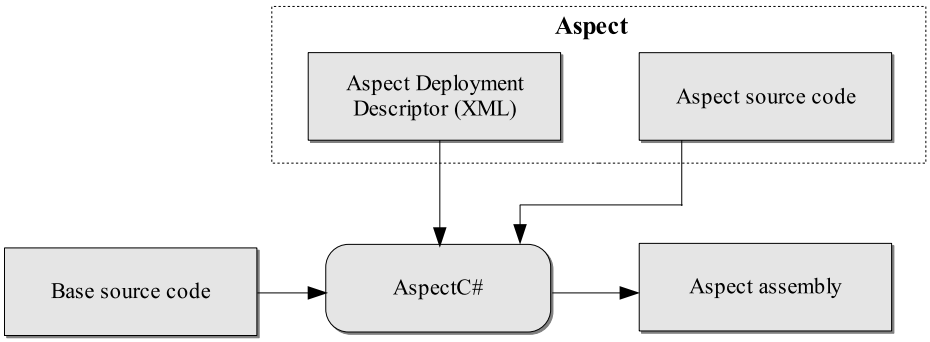


Figure 4.1: Architecture of AspectC#.

One of the interesting aspects of AspectC# is the `Aspect Deployment Descriptor`, which among other things contains the information on which the base and source code are located together with the well defined join points. Below is illustrated an example of how such an `Aspect Deployment Descriptor` looks like:

```
<?xml version="1.0" encoding="utf-8" ?>
<Aspect>
  <TargetBase>C:\Project\MoveTracking</TargetBase>
  <AspectBase>C:\Project\MoveTrackingAspect</AspectBase>
  <Aspect-Method>
    <Name>AspectBefore</Name>
    <Namespace>MoveTracking</Namespace>
    <Class>MoveTracking</Class>
    <Method>before()</Method>
  </Aspect-Method>
  <Target>
    <Namespace>Test</Namespace>
    <Class>HelloWorld</Class>
    <Method>
      <Name>SetX</Name>
      <Type>before</Type>
      <Aspect-Name>AspectBefore</Aspect-Name>
    </Method>
    <Method>
      <Name>SetY</Name>
      <Type>before</Type>
      <Aspect-Name>AspectBefore</Aspect-Name>
    </Method>
  </Target>
</Aspect>
```

The `TargetBase` and `AspectBase` represent the location of the C# source code and Aspect code. An `Aspect-Method` is an aspect method and consists of a name, namespace of the class, class name and the method name. A `TargetMethod` is the target or base class. It consists of namespace, class name and method name.

When designing AspectC# it was decided to not extend the C# language, so the standard .NET C# compiler can be utilized to compile the woven source code into the final .NET assembly.

4.3 Weave.NET

Weave.NET [17] is an ongoing project for applying aspects to any language that compiles into a CLI assembly. Weave.NET uses an XML document to describe the crosscutting concerns that are to be captured in the CLI assembly and the user must supply an additional CLI assembly which contains the implementation of how to act on each of the crosscutting concerns. To illustrate an XML document in Weave.NET a subset of such is illustrated below:

```
<ax:aspect>
<name>Point</name>
<assembly>Point_Assembly</assembly>
<type>MoveTracking</type>
<body>
<item><advice><before>
   <formal_param>
      <var_Type>object</var_type>
      <var_name>data</var_name>
   </formal_param>
   <pointcut><primitive><pointcutId>
      <name>Move</name>
   </pointcutId></primitive></pointcut>
   <behaviour>
     <name>LogWrite</name>
   </behaviour>
</before></advice></item>
...
...
...
</body>
</ax:aspect>
```

The XML document is structured much like an aspect in AspectJ and supports many of the same structures like advices, pointcuts, etc. except in Weave.NET tags are used instead of code. First the name and the assembly defining the behavior of the aspect is defined, where after the body is defined which e.g. could be advices.

The weaving process is then performed by manipulating the source assembly's metadata and executable code such that the user-supplied assembly is called for

each of the defined crosscuts. Figure 4.2 illustrates the overall architecture of Weave.NET.

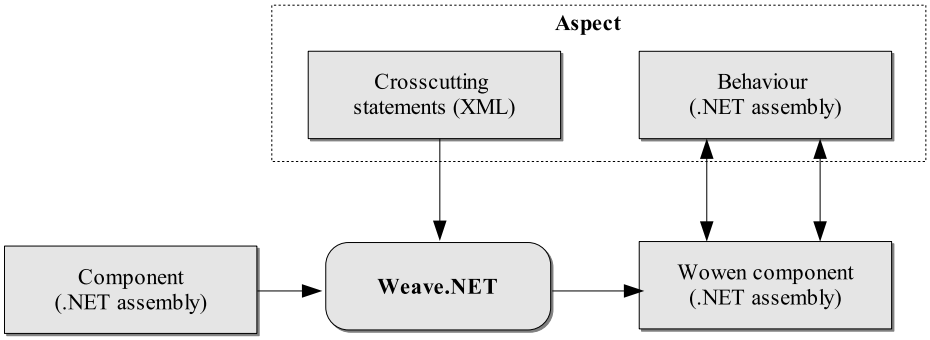


Figure 4.2: Overview over the Weave.NET system.

Weave.NET offers a subset of the features that exist in AspectJ (see section 4.1) - e.g. the `before` and `after` advice. Table 4.1 and 4.2 provides an overview over which features that are and are not supported by Weave.NET.

Pointcut Designators	**Supported**
call	yes
execution	yes
get	no
set	no
handler	no
initialization	no
staticinitialization	no
within	yes
withincode	no
cflow	no
cflowbelow	no
this	yes
args	yes
target	no

Table 4.1: Primitive pointcut designators support currently implemented in Weave.NET.

As mentioned, before Weave.NET offers a subset of AspectJ although some of the more important and "powerful" features of AspectJ are not supported. From Table 4.1 and 4.2 it can be seen that Weave.NET e.g. do not offer support for: `cflow`, `target` and the `around` advice.

Advice	Supported
Before	yes
Around	no
After	yes
After returning	yes
After throwing	no

Table 4.2: Advice support currently implemented in Weave.NET.

4.4 A dynamic AOP-Engine for .NET

Dynamic weaving is the ability to allow weaving and un-weaving at runtime. The dynamic AOP-engine for the .NET platform [13] is a dynamic aspect weaver which contains the basic functionality, namely that aspects are executed before, around and after a given method. The dynamic AOP-engine uses a so-called AOP Debugger which has been designed for the CLR's debugging interface to perform weaving and un-weaving at runtime. The AOP Debugger is based on the debugger which is made available in Microsoft's Shared Source Common Language Infrastructure[2] 1.0 (SSCLI). The API of the CLR's debugging interface provides functionality to access metadata and the current state of a process, which is used to load and unload aspects. Given that the AOP Debugger is based on the debugger from SSCLI makes the AOP Debugger platform-dependent.

In order to weave aspects at runtime, it must be possible to let the running program know when to weave aspects. A simple way to do this could be to insert breakpoints into the running program, however, in .NET when reaching a breakpoint all threads are suspended before continuing execution and this would cause the entire application to stop. Even though standard breakpoints are not a possibility, the CLR debugging mechanism can still be used, it just need to be extended to support insertion of aspects. Therefore it has been decided to insert stubs into the method bodies of the running program. Then whenever the stub is reached, the control is switched over to the AOP Debugger. When the desired action has been performed, e.g. insertion of an aspect, the control is switched back to the running program.

This dynamic AOP-engine accepts aspects written in any .NET language as long as it is compiled to a dynamic link library (DLL). Furthermore the DLL files must as a minimum contain a class which implements a specific aspect interface.

[2] http://msdn.microsoft.com/net/sscli

Part II

Design

This part investigates different design approaches for designing a cross language aspect weaver and concludes which of these is best suited for the design of Aspect.NET. The syntax and semantics for the Aspect.NET language are then introduced along with examples of how to capture and implement crosscutting concerns.

Chapter 5

Capturing aspects in .NET languages

When you're as great as I am, it's hard to be humble.

— Muhammed Ali

This chapter investigates different approaches for designing an aspect-oriented extension for .NET languages that will support aspect weaving regardless of language. The design of this aspect-oriented extension is focused on giving .NET users an AspectJ-like language and compiler which supports dynamic crosscutting by using compile-time (or static) aspect weaving.

There are, however, several difficulties in designing such a tool because of the variations that exist in programming languages. Therefore the first section of this chapter starts by outlining the challenges that one is facing when designing a cross language aspect-oriented extension. The second section of the chapter investigates different design approaches that will make it possible to provide users with a common way of intercepting calls to what is termed generalized-procedures in programming languages. The last section will then conclude which of these design approaches will be used for the design of Aspect.NET.

5.1 Design considerations

There are many ways to weave aspects into a program. The dominant AOP-tool AspectJ is designed as an extension to the Java language which makes aspects a quite natural part of the program, since both the program and the aspects that are captured in the program uses the same language compiler. This also allows the aspect extension to make full use of the features that are available in the original language. In Aspect.NET the goal is, however, to find a common way to capture crosscutting concerns in a wide range of languages. Common to all languages for which AOP applies is that they have some kind of generalized-procedure which may represent a crosscutting concern and therefore

Aspect.NET must be able to identify and locate calls to generalized-procedures. The generalized-procedures must, however, be found in the compiled code of .NET language compilers, namely the CIL.

Providing aspect weaving to all .NET languages by using a program's compiled code is however quite a challenge because once a program is compiled to CIL, it may be hard to map the compiled program to its original structure from its high-level language. One of the reasons for this is that CIL is based upon an object-oriented model for expressing programs which makes it rather simple to map the structure of an object-oriented program to CIL, while a program written in e.g. a functional language has to make use of tricks in order to be mapped to the object-oriented model used for CLI assemblies. In other words object-oriented .NET languages like C#, Managed C++, J#, JScript and Visual Basic[1] are mapped nicely to assemblies and to a great extent reflect their high-level languages. While a functional language like F# [25] from the ML-family, has to make use of object-oriented features in order to be compiled to a CLI assembly although it is not an object-oriented language.

To give a simple example of this consider the following F# program which has a function that adds two numbers:

```
let add x y = x + y
```

Once compiled to CIL the above program looks as follows (not including the method bodies):

```
.class public auto ansi beforefieldinit Add
       extends [mscorlib]System.Object
{
  .method private specialname rtspecialname static
          void  .cctor() cil managed
  { ... } // end of method Add::.cctor

  .method public static int32  'add'(int32 x,
                                     int32 y) cil managed
  { ... } // end of method Add::'add'

} // end of class Add
```

From this code-snip it is clearly shown how the F# compiler uses the object-oriented model in CIL in order to compile the F# program. In this case the F# compiler has created a class in the assembly which has the name of the source file being compiled and the `add` function in the F# program has been created as a static method to that class.

More than 30 languages are already available for .NET (listed in appendix D) and the compilers for each of those languages generate CIL code in very different ways. An aspect weaver that supports all these languages is a demanding task and initially the design of Aspect.NET therefore only investigates how a subset of these languages can be supported. These languages are the following: C#

[1] All languages part of Microsoft Visual Studio .NET

[21], J# [29], JScript [26], Managed C++ [27], C++ [4], Visual Basic.NET [22], Eiffel.NET [33], SML.NET [2], F# [25] and Component Pascal [32]. The next three subsections investigate some of the challenges that are involved in designing a cross languages aspect weaver for these languages.

5.1.1 Code mangling

The main reason that some compilers produce CLI assemblies with mangled CIL code is that the languages they are compiling, contains language constructs that are directly not supported by CIL, and the compiler therefore needs to generate additional code for supporting these language constructs. The following subsections investigate some of these language constructs and how they will complicate the search for generalized-procedures in the generated CIL code.

Multiple inheritance in C++ and Eiffel.NET

Managed C++ does not allow multiple inheritance, but "standard" C++ and also Eiffel.NET allow multiple inheritance which is a not directly supported by CIL. Still both languages allow use of multiple inheritance on the .NET platform. The Microsoft C++ compiler implements multiple inheritance in CIL by creating each class as a value type and the methods of each class then becomes global functions where the first parameter is a pointer to its value type. Although the C++ compiler thereby simulates multiple inheritance in CIL, finding the inheritance pattern between classes becomes difficult.

Multiple inheritance in Eiffel.NET is simulated in a more CLS-friendly manner by using multiple interfaces, each of which has a so-called shadow class that contains the implementation. A more detailed description is available at MSDN[2]. Finding the inheritance pattern and mapping to the high-level language from CIL is therefore a bit simpler in Eiffel.NET than it is in C++.

Nested functions in Component Pascal

Component Pascal also has language constructs that is not directly supported by CIL and therefore also resorts to tricks in order to allow these language constructs in CIL. One example are nested functions which are functions within functions where a nested function can access local variables in all its parent functions. Here is an example of two nested functions that are members of a module:

```
PROCEDURE First();
VAR a : INTEGER;
   PROCEDURE Second();
   VAR b : INTEGER;
      PROCEDURE Third();
      VAR c : INTEGER;
      BEGIN
```

[2] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp

```
        c := a+b; Console.WriteString("Result: "); Console.WriteInt(c,0);
      END Third;
   BEGIN
     b := 2;
     Third();
   END Second;
BEGIN
  a := 1;
  Second();
END First;
```

Once the above example is compiled to CIL, the functions are represented in the following way:

```
.method private static void 'Third@Second@First'(class [RTS]XHR) il managed
.method private static void 'Second@First'(class [RTS]XHR) il managed
.method private static void 'First'() il managed
```

The Component Pascal compiler simulates nested functions by creating a new method for each function and then passing the variables of the parent functions onto nested functions through an object of class `XHR`.

Functional-like constructs

Language constructs that are far from the object-oriented model used by CIL, like e.g. those that are provided by functional languages, present quite a challenge to those who have to implement the compilers for .NET. One example are high-order functions for which CIL has no support. Both F# and SML.NET support high-order functions but the way this language construct has been implemented in the compilers of these two languages is quite different.

F# has been designed with basis on ensuring language interoperability and therefore makes use of a library called ILX [34] which provides better support for functional language constructs. The mapping from a program written in F# to its compiled version in CIL is therefore still possible to analyze. The implementors of the SML.NET compiler, on the other hand, seems to have done very little to ensure language interoperability. Once a language construct is used which is not directly supported by CIL, the generated CIL code which represents this language construct becomes almost impossible to analyze.

To illustrate how F# maintains a clear mapping to the original program during compilation, consider the following F# program:

```
let add (a, b) = a + b
let hfun f x y = f(x, y)
let _ = print_int ( hfun add 4 4 )
```

Once compiled to CIL, the compiler uses the file name of the .ml file as the class name for encapsulating the functions that are defined in the F# program. Here is a code-snip of the generated CIL code:

```
.class public auto ansi beforefieldinit High
       extends [mscorlib]System.Object
{
  .method public static int32 'add'(int32 a,
                                     int32 b) cil managed
  {...}
  .method public static object hfun(class [ilxlib]System.Func1 f,
                                     object x,
                                     object y) cil managed
  {...}

  .method public static void main() cil managed
  {.entrypoint ...}
}
```

Above is shown how the function `add` and the high-order function `hfun` are traceable in the compiled version. The high-order function, however, uses arguments from the ILX library in order to support this language constructs. But still the mapping from functions in the original program are visible after compilation and can therefore be intercepted once they are called.

If the above F# program is written for SML.NET, it looks as follows:

```
structure High :>
sig
  val main : unit -> unit
end
= struct

  fun add (a, b) = a + b
  fun hfun f x y = f (x, y)
  fun main() = print(Int.toString( hfun add 4 4 ))
end
```

Once compiled to CIL finding the functions in the above program is almost impossible because the SML.NET compiler does not maintain identification of functions. Below are code-snips of the above program once it has been translated by the SML.NET compiler. Note that the CIL code has only been included in order to emphasize the degree of code mangling that occur during compilation to CIL.

```
.namespace $
{
  .class private abstract auto ansi Class_a
         extends [mscorlib]System.Exception
  {
    .method public specialname rtspecialname
            instance void  .ctor(string A_0) cil managed
    { ...} // end of method Class_a::.ctor
```

```
    .method public virtual instance string
            ToString() cil managed
    { ... } // end of method Class_a::ToString

    .method public virtual instance string
            ExnMessage() cil managed
    { ... } // end of method Class_a::ExnMessage

    .method public virtual instance string
            ExnName() cil managed
    { ... } // end of method Class_a::ExnName

  } // end of class Class_a

  .class private auto ansi Fun
         extends [mscorlib]System.Object
  {
    .method assembly specialname rtspecialname
            instance void  .ctor() cil managed
    { ... } // end of method Fun::.ctor
  } // end of class Fun

  .class private auto ansi Globals
         extends [mscorlib]System.Object
  {
    .method assembly static void  $() cil managed
    { ... } // end of method Globals::$
  } // end of class Globals
} // end of namespace $

.class public auto ansi sealed High
       extends [mscorlib]System.Object
{
  .method public specialname rtspecialname static
          void  .cctor() cil managed
  { ... } // end of method High::.cctor

  .method public static int32  main() cil managed
  { ... } // end of method High::main
} // end of class High
```

5.1.2 Renaming of identifiers

When designing a cross language aspect weaver, it is very important that compilers maintain the naming of identifiers during compilation, because AOP's main focus is on calls to generalized-procedures. The previous section has already shown examples of how generalized-procedures have their names changed in order to support e.g. nested functions in Component Pascal. Here the mapping to CIL is however quite clear and simple to map to programs written in

the high-level language.

In some languages renaming of identifiers occurs intentionally in order to promote cross language interoperability, and identifiers for types, methods and fields in CIL therefore no longer have a direct mapping to programs written in the high-level languages. Eiffel.NET is an example of a language that has a compiler which renames identifiers. A method on a type like e.g. `BUTTON.IS_ENABLED` will, once compiled, be renamed to `Button.IsEnabled()` for better support of language interoperability.

5.1.3 Mixed mode assemblies

Not all .NET compilers produce assemblies that are entirely based on managed code. The reason for this is that parts of a program may be compiled to native code which is not verifiable by the CLR. Manipulation of such assemblies is not supported by the .NET framework and therefore cannot be used for aspect weaving. The Microsoft C++ compiler is so far the only compiler that has been found to produce mixed mode assemblies.

5.2 Possible design approaches

When designing a cross language aspect weaver, the most important factor is being able to identify calls to so-called generalized-procedures which represent crosscutting concerns. However, as described in section 5.1, once a program is compiled to CIL it is, in some cases, quite difficult to find the mapping between the compiled code and the program in the high-level language.

In most cases the object-oriented language compilers generate CIL code which is possible to reverse engineer and easily locate generalized-procedures, especially the Microsoft's .NET languages C#, J#, JScript, Managed C++ and Visual Basic.NET. Programs written in C++, Eiffel.NET and Component Pascal are also possible to reverse engineer although the mapping from CIL to the original program is a bit more challenging. Also F# showed to be a bit challenging although the mapping to the original program is still traceable. The really tough challenge comes with a language like SML.NET where the compiler generates CIL code that has little or no mapping to generalized-procedures in the original program.

Given that the CIL code being produced by the different language compilers varies greatly because of language constructs that are not directly supported by CIL, complicates the design of an aspect-oriented language that uniformly will allow users to capture calls to generalized-procedures. Therefore two design approaches are seen as most fit for allowing an AspectJ-style approach for weaving aspects across .NET languages. Either a low-level approach that entirely focuses on the CIL code generated by the language compiler, or a more high-level approach that focuses on languages which are closely related to the object-oriented model available in CIL. These two approaches are investigated in more detail in the two following sections.

5.2.1 An AOP-language for CIL

Designing an AOP-language that is based on supporting language constructs available in CIL makes it possible to support aspect weaving for all .NET languages since they all compile to CIL. The AOP-language used in such an approach is then based on capturing the different types of generalized-procedures that are available in the object-oriented model used for CLI assemblies, namely global methods and methods that are members of classes.

The downside is, however, that the user has to be familiar with the way that his language compiler translates a program to CIL code. So if the user wishes to introduce an aspect which intercepts all calls to a function `add` in e.g. a F# program, then in order to capture this aspect the user needs to investigate the assembly that the F# compiler outputs and locate the `add` function, e.g. by using a CIL disassembler tool like IL DASM or Visual Studio's Object Browser. Figure 5.1 illustrates how IL DASM can be used to investigate an assembly generated by the F# compiler.

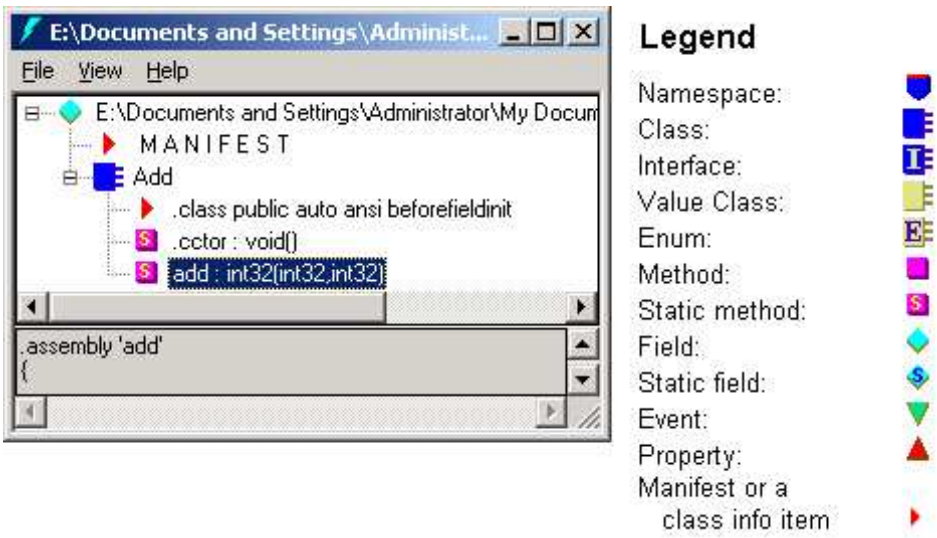


Figure 5.1: IL DASM showing a compiled version of a F# program containing a function `add` which adds two integers.

5.2.2 An AOP-language for closely related languages

Since the mapping to a program written in a high-level language may be lost during compilation or be very hard to analyze, the design of a AOP-language can instead be focused on only supporting a limited number of languages which have similar language constructs and a clear mapping to CIL once compiled. This essentially means that the AOP-language will only support C#, Visual Basic.NET, Managed C++ and JScript, since the CIL code produced by these language compilers is simple to reverse engineer to the original programs.

Although this greatly limits the number of languages that can be used for cross language aspect weaving, it will provide users with an AOP-language that has better support for the language constructs that can represent crosscutting concerns in these languages.

5.3 Approach for Aspect.NET

The two design approaches described in section 5.2 are very different in the way they will support cross language aspect weaving. If choosing to design an AOP-language that is based on the compiled code of programs, namely CIL, then the high-level language that is used is completely disregarded. The second approach, on the other hand, will maintain the mapping to the high-level language, such that the user does not have to be familiar with CIL, but this will limit the number of supported languages.

The design of Aspect.NET is based on the approach which provides users with an AOP-language for capturing crosscutting concerns in CIL. Although an AspectJ-style language may seem odd for a low-level language like CIL, it is in fact well suited for capturing crosscutting concerns. The reason for this is that AspectJ encapsulates the implementation of crosscutting concerns in a object-oriented fashion which is well suited for CIL's way of expressing programs. Some modifications must however be made to the syntax used by Aspect.NET such that it fits the terms used in CIL instead of the Java language. One example is e.g. to use the term namespace instead of package. In addition, Aspect.NET supports a wider range of generalized-procedures than those supported by AspectJ because CIL has support for both methods that are members of types and global methods.

Once a user has specified a crosscutting concern, the Aspect.NET language must also provide the user with a way to implement this crosscutting concern (in AspectJ a so-called advice) and since Aspect.NET is a cross language aspect-oriented extension, the implementation of a crosscutting concern should not be restricted to a single language. Aspect.NET therefore allows users to write the implementation of crosscutting concerns in the language of their choice. This is possible either by writing the implementation within the aspect definition or by referring to a static method in another assembly. If the implementation of a crosscutting concern is supplied in a separate assembly the only restriction on the method that encapsulates this concern is that it is static and returns void such that the evaluation stack is left untouched.

Since Aspect.NET allows cross language aspect weaving, it actually means that the implementation of a crosscutting concern can use a language that is different from the language used for implementing the program targeted for aspect weaving. This gives users the ability to implement functionality in crosscutting concerns that may not be provided by the language used for implementing the program targeted for aspect weaving.

Chapter 6

The Aspect.NET language

Premature optimization is the root of all evil.

— Donald Knuth

This chapter outlines the language constructs that are available in the Aspect.NET language for modularizing crosscutting concerns. The syntax and semantics of the Aspect.NET language are introduced by using examples which illustrate how to capture and implement crosscutting concerns. Appendix A provides a BNF-grammar of the possible derivations in the Aspect.NET language.

The examples used for outlining the features available in Aspect.NET are based on the sample code in appendix E which simulates a simple graphical display. The UML diagram for the graphical display is illustrated in Figure 3.1 of chapter 3.

6.1 Syntax and semantics of Aspect.NET

This section introduces the syntax used in the Aspect.NET language and the semantics of each language construct. For a complete list of the keywords used in Aspect.NET refer to appendix B and the grammar used by Aspect.NET is described in appendix C.

Before going into detail on each of the language constructs, a few global language constructs will be outlined. Aspect.NET is a case sensitive language and supports commenting by using both the `//` for a single line and `/*` ending with `*/` for several lines. Identifiers must use the following regular expression: $[a-zA-Z\_\$@?][a-zA-Z0-9\_\$@?]*$

A source file for the Aspect.NET compiler must have the file extension .dna which is an abbreviation for “dot NET aspect”.

6.1.1 Defining aspects

Aspect.NET supports definition of aspects in a similar way as classes are defined in widely used OOP languages. The following example illustrates the definition of an aspect which will be named `Tracer`:

```
aspect Tracer
{
   /*
    * Comments...
    */
   // More comments...
}
```

6.1.2 Name separation

In order to make naming of aspects more flexible Aspect.NET supports namespaces which can incorporate a set of aspects. If an aspect requires types from other namespaces these can also be included. Here is an example of an aspect named `Tracer` being declared in a namespace called `Trace` which requires that the `Graphics` namespace is included:

```
using Graphics;

namespace Trace
{
   aspect Tracer
   {
      /* ... */
   }
}
```

Aspect.NET also supports sub-namespaces such that a namespace can be defined within another namespace. The separation of namespaces is indicated by a dot (“.”).

6.1.3 Pointcuts

Aspect.NET is able to capture crosscutting concerns in both named and unnamed pointcuts. A pointcut consists of one or more pointcut designators which are defined by the `call` keyword followed by a return type, a type and a method. If multiple pointcut designators are specified, they can be separated by the conditional-OR operator ||. The following example illustrates an aspect which has a named pointcut that captures movement of a graphical component:

```
using Graphics;

namespace Trace
```

```
{
   aspect Tracer
   {
      pointcut Movement() : call ( void Line.MoveBy(int, int) );
   }
}
```

If the parameters for a join point are being passed as reference, the types should be preceded by the `ref` keyword. If a parameter is an array then the type should be followed by `[]`. A join point in Aspect.NET can also be a method on a nested type, a global method and a constructor. The following three pointcuts give examples of intercepting calls to each of these join points:

```
/* Intercept calls to a method on a nested type */
pointcut InterceptNested() : call ( void MyClass.Nested.Foo(int) );

/* Intercept calls to a global method */
pointcut InterceptGlobal() : call ( int MyGlobalMethod( int ) );

/* Intercept calls to a constructor */
pointcut InterceptConstructor() : call ( void MyClass..ctor( int ) );
```

Static weaving and dynamic binding

An important factor to keep in mind when using static weaving, which is the case in Aspect.NET, is that in some languages method invocations are not determined at compile time but instead at runtime. To illustrate the problem this imposes, consider the following three classes written in C# and an aspect written in Aspect.NET:

```
class A
{
   public virtual void foo(){}
}

class B : A
{
   public override void foo(){}
}

class C
{
   public static void bar(A a)
   {
      a.foo();
   }
}

aspect Intercept
```

```
{
   pointcut register() : call ( void B.foo() );
}
```

Note that the class `B` is derived from the class `A` and that `B` overrides the method `foo()` of `A`. The aspect defined as `Intercept` then intercepts calls to the method `foo()` on `B`. If, however, an object of class `B` is passed to the method `bar(A a)` of class `C` and the method `foo()` is invoked on this object, then this call will not be intercepted by the aspect `Intercept` although there is a pointcut on the `foo()` method of `B`. The reason is that at compile time when aspects are woven there is no way to determine the type of the object being passed to `C.bar(A a)`.

6.1.4 Pointcuts in a particular context

A user may not always wish to intercept calls to a join point unless the call occurs in a certain context. Aspect.NET therefore allows users to check the current control flow by using the `cflow` keyword which can also be used in combination with the logical negation operator "!". The following example shows a pointcut which detects when a `Point` is moved, but only if it occurs when a `Line` is being moved:

```
using Graphics;

namespace Trace
{
   aspect Tracer
   {
      pointcut LineMovement() :
               call ( void Point.MoveBy(int, int) ) &&
               cflow ( call (void Line.MoveBy(int, int) ) );
   }
}
```

`cflow` investigates the call-stack at runtime in order to check the current context and therefore imposes a slight performance penalty. Note that `cflow` is **not** thread-safe.

6.1.5 Accessing parameters

The parameters that are being passed to the method specified in the pointcut designator can also be included for later use by specifying the `args` keyword and using the conditional-AND operator && for mapping it to a method. The naming of the pointcut must then also include variables which will be assigned values of the parameters passed to the method. The following example illustrates how:

```
using Graphics;
```

```
namespace Trace
{
   aspect Tracer
   {
      pointcut Movement(int dx, int dy) :
               call ( void Line.MoveBy(int, int) ) &&
               args(dx,dy);
   }
}
```

The above example intercepts all calls to `void Line.MoveBy(int, int)` and stores the values of the two parameters passed to the method in variables `dx` and `dy` specified in the declaration of the pointcut. If the parameters for a join point are being passed as reference, the variables in the pointcut definition should be preceded by the `ref` keyword and likewise in the method definition in the join point. Aspect.NET also allows the use of call-by-reference such that users can manipulate the arguments that will be passed to a join point. Section 6.1.7 shows an example of the use of call-by-reference.

6.1.6 Accessing objects

If the pointcut designator describes a method that is a member of an object then this object can be accessed by specifying the `target` keyword. If `target` is specified, a variable of that type must also be specified in the naming of the pointcut. The following example shows the use of `target` keywords:

```
using Graphics;

namespace Trace
{
   aspect Tracer
   {
      pointcut Movement(Line line) :
               call (void Line.MoveBy(int, int)) &&
               target(line);
   }
}
```

The above example makes the object that calls the `Line.MoveBy(int, int)` accessible in the variable `line` specified by the `target` keyword for later use. Note that `target` will return false if the method specified in the join-point is either a static member of a class or a global method.

6.1.7 Advices

An advice is an implementation of a crosscutting concern that will be executed when a pointcut is intercepted. Aspect.NET supports both before-, after- and around advice which means that an advice can be executed before and after

a method is called or the called method can be replaced by using an around advice. The advices can be implemented in two ways, either by writing the advice within the aspect definition or by referring to a method of a type in a CLI assembly. If the implementation of the advice is supplied by the user in an assembly, then it must be implemented in a static method. The CLI assembly containing the advice implementation can be either a separate assembly or it can be the same assembly that is also targeted for aspect weaving. In both cases the implementation of advices are copied into the woven assembly and the new assembly will therefore have no external references to those specified in the aspect definition.

The following example shows how an advice supplied in a CLI assembly can be referred to from the Aspect.NET language. The example is designed to check that a `Line` will not be moved out of the display area using a `before`-advice and once a `Line` has been moved the screen will be updated using an `after`-advice. An unnamed pointcut for an `around`-advice is also used to redirect console output to a file instead.

```
using Graphics;
using System;

namespace Trace
{
   aspect Tracer
   {
      pointcut Movement(Point point, int dx, int dy) :
               call ( void Point.MoveBy(int, int) ) &&
               cflow ( call (void Line.MoveBy(int, int) ) ) &&
               target(point) && args(dx, dy);

      before(Point point, int dx, int dy): Movement(point,dx,dy)
        [Graphics.exe, Trace.Aspect.VerifyPoint(point, dx, dy)]

      pointcut LineMoved() : call ( void Line.MoveBy(int, int) );

      after(): LineMoved()
        [Graphics.exe, Graphics.Display.Update()]

      around(string str) : call ( void Console.WriteLine(string) )
                           && args( str )
        [Graphics.exe, Trace.Aspect.WriteLog(str)]
   }
}
```

The above examples shows that the implementation of advices, in all cases, is located in an assembly called Graphics.exe followed by the namespace-path to the class and method which contains the advice implementation.

One can also choose to write the implementation of advices within the aspect definition by encapsulating the advice implementation with the tags <? and ? >,

similar to how PHP-scripts[1] separates HTML from PHP. The language used for implementing the advice directly follows the starting <? by its language ID. So if the advice is implemented using C# which has the language ID C# it's written like this: <?C# ... ? >. The example above looks as follows if the implementation of advice is written within the aspect definition:

```
using System;
using System.IO;
using Graphics;

namespace Trace
{
   aspect Tracer
   {
      pointcut Movement(Point point, int dx, int dy) :
               call ( void Point.MoveBy(int, int) ) &&
               cflow ( call (void Line.MoveBy(int, int) ) ) &&
               target(point) && args(dx, dy);

      before(Point point, int dx, int dy) : Movement(point,dx,dy)
         <?C#
              if( point.GetX()+dx >= Display.GetWidth() ||
                  point.GetY()+dy >= Display.GetHeight() ||
                  point.GetX()+dx < 0 || point.GetY()+dy < 0 )
                  throw new Exception ("Line will be out of bounds");
         ?>

      pointcut LineMoved() : call ( void Line.MoveBy(int, int) );

      after(): LineMoved()
         <?C#
               Display.Update();
         ?>

      around(string str) : call (void Console.WriteLine(string) )
                           && args (str)
      <?C#
        StreamWriter sw = new StreamWriter("Logfile.log", true);
        sw.WriteLine(str);
        sw.Close();
      ?>
   }
}
```

If the implementation of advices is written within the aspect definition, the Aspect.NET compiler will extract the code between the starting <? and ending ? > and compile this separately by the corresponding compiler. Section 6.1.8 has a list of the languages which are possible to use for inline implementation of advices.

[1] http://www.php.net

Call-by-reference

Aspect.NET allows the use of call-by-reference on arguments that are being passed to a join point. Users can thereby change the value of the arguments that are e.g. used in a before advice. To illustrate how call-by-reference is used, consider the example in the previous section. Instead of throwing an exception when a call to `MoveBy` is made that will send a `Line` object out of bounds, call-by-reference can be used to draw the `Line` within bounds. Here is how (the around-advice has not been included):

```
using Graphics;

namespace Trace
{
   aspect Tracer
   {
      pointcut Movement(Point point, int dx, int dy) :
               call ( void Point.MoveBy(int, int) ) &&
               cflow ( call (void Line.MoveBy(int, int) ) ) &&
               target(point) && args(dx, dy);

      before(Point point, ref int dx, ref int dy) : Movement(point,dx,dy)
         <?C#
              if( point.GetX()+dx >= Display.GetWidth() )
                dx = Display.GetWidth() - point.GetX();
              if( point.GetY()+dy >= Display.GetHeight() )
                dy = Display.GetHeight() - point.GetY();
              if( point.GetX()+dx < 0)
                dx = point.GetX() * (-1);
              if( point.GetY()+dy < 0 )
                dy = point.GetY() * (-1);
         ?>

      pointcut LineMoved() : call ( void Line.MoveBy(int, int) );

      after(): LineMoved()
         <?C#
               Display.Update();
         ?>
   }
}
```

The above example checks that if a point is being moved out of the display's bounds within the context of a call to `Line.MoveBy`, then the point is set to either the display's minimum or maximum value. The use of call-by-reference is indicated in the variables used in the before advice where `dx` and `dy` have been preceded by the `ref` keyword.

An important concept when using call-by-reference on arguments that would normally be passed by value is that if an argument's value is changed, the

change is only visible to the advices and the join point. This is best illustrated by an example:

```
public class MyClass
{
  public static void Foo(int a)
  {
    a++;
    System.Console.WriteLine("Foo exit, a="+a);
  }

  public static void Main()
  {
    int a = 0;
    Foo(a);
    System.Console.WriteLine("Main exit, a="+a);
  }
}

aspect Intercept
{
  pointcut Change(int a) :
   call (void MyClass.Foo(int) ) && args(a);

   before (ref int a) : Change(a)
   <?C#
    a++;
    System.Console.WriteLine("Before Foo, a="+a);
   ?>

   after (ref int a) : Change(a)
   <?C#
    System.Console.WriteLine("After Foo, a="+a);
   ?>
}
```

If `MyClass` is executed without the aspect `Intercept`, then the output will be the following:

```
Foo exit, a=1
Main exit, a=0
```

If `MyClass` is executed with the aspect `Intercept`, then the output is instead:

```
Before Foo, a=1
Foo exit, a=2
After Foo, a=1
Main exit, a=0
```

The above example shows that Aspect.NET ensures that value types which are passed by reference to an advice will only appear changed to the advice

implementations and their join points, but not for the later execution. This design choice is helpful in maintaining the current state in the method calling the advice. Consider e.g. the aspect in the previous example where call-by-reference was used to keep a `Line` object within the display area. If instead there were two `Line` objects, `l1` and `l2`, which must follow each other then not maintaining the current state of the method's local variables would affect later execution. The following example shows how:

```
static void Foo(Line l1, Line l2)
{
  int a = 50, b = 50;
  l1.MoveBy(a,b);
  l2.MoveBy(a,b);
}
```

If the call to `l1.MoveBy(a,b)` makes `l1` go outside the display area and a before-advice manipulates the input arguments such that `a` and `b` are manipulated in the `Foo`-method as well, then these changes would affect the movement of `l2` such that it would no longer follow `l1`. When call-by-reference only affects the advice implementation and the join-point, which is the case in Aspect.NET, `l2` will still be passed with the arguments where $a = 50$ and $b = 50$.

6.1.8 Inline advice implementation

So far Aspect.NET only supports a limited number of languages for implementation of advices within the aspect definition. The list of supported languages is listed in Table 6.1.

Language	**Language ID**
C#	C#
F#	F#

Table 6.1: Language ID.

6.1.9 Types

To ease implementation, Aspect.NET supports a limited number of primitive types when specifying parameters for pointcuts. The types supported by Aspect.NET are listed in Table 6.2 along with their corresponding CTS type.

Aspect.NET type	**CIL type**
sbyte	int8
short	int16
int	int32
long	int64
byte	unsigned int8
ushort	unsigned int16
uint	unsigned int32
ulong	unsigned int64
float	float32
double	float64
bool	bool
char	char
IntPtr	native int
UIntPtr	native unsigned int
object	object
string	string

Table 6.2: Mapping between Aspect.NET types and types in the CIL.

Part III

Implementation

This part describes the implementation of the Aspect.NET compiler and outlines how each of the more advanced language constructs are implemented.

Chapter 7

Overview

The hardest work in the world is that which should have been done yesterday.

— Anonymous

Once a user has written an aspect definition in the Aspect.NET language and created an assembly containing the implementation of advices, the actual weaving of aspects into an existing software component can begin. This chapter gives an abstract view of the core components in the Aspect.NET compiler and also introduces the tools that will be used in the development of the compiler.

7.1 The Aspect.NET compiler

The Aspect.NET compiler consists of a lexer, a parser and an aspect weaver. The lexer scans through the stream of characters of the aspect definition written in the Aspect.NET language and creates a list of tokens. This list of tokens is then transformed into a tree structure by the parser and is ready to be traversed for the defined pointcuts and corresponding advices. The aspect weaver is then initiated and starts the weaving process by creating a clone of the original assembly. The CIL instructions of the cloned assembly are then scanned for calls to the set of pointcuts that are defined in the Aspect.NET program. If a call is intercepted to a pointcut, its corresponding advice is then weaved into the cloned assembly and the CIL instructions are modified to include execution of the advice. The implementation of advices for each pointcut can be provided by the user in a separate assembly or inlined in the aspect code. Figure 7.1 gives an abstract view of the components and steps involved in the weaving process.

7.2 Compiler support tools

To support the development of the Aspect.NET compiler, an existing compiler-compiler tool called CSTools 4.5 [7] is used to generate the lexer and parser. The weaving process mainly uses the .NET class library and especially the classes

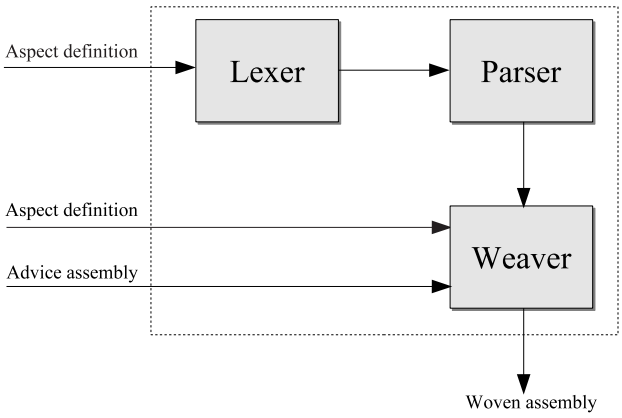


Figure 7.1: Abstract view of the weaving process.

`System.Reflection` namespace. The .NET class library, however, only supports loading of assemblies and not manipulation of the loaded assembly, so it is e.g. not possible to modify a type or investigate a method's instruction list. The weaving process must therefore start by creating a clone of the original assembly by using the classes in the `System.Reflection` namespace. The classes in `System.Reflection`, however, does not allow user' access to the executable code of methods so a tool called IL Reader is used for this task. The following three sections gives a more detailed description of the tools used by the Aspect.NET compiler.

7.2.1 CSTools 4.5

Instead of building a compiler from scratch, compiler writing tools have been developed which do most of the work for the user. CSTools 4.5 is one such tool which is developed by Malcolm Crowe and documented through Compiler Writing Tool Using C# (CWTUC#). CWTUC# is able to automatically generate the lexer from a set of rules, which must be specified by the user. It can also generate a parser from a pre-defined syntax. The necessary rules for the lexer and the necessary syntax for the parser can be written in any standard available text editor. Hereafter the lexer generator (lg) and the parser generator (pg) is used with the text files as input. The result of successfully running lg and pg are two C# source files containing a lexer and a parser. The parser returned from using pg is an LALR parser (Look-Ahead Left to Right parsing).

7.2.2 Libraries for IL generation

To create a new assembly, the .NET framework offers an API called `System.Reflection` which is used to handle assemblies and its types, methods and fields. The API includes a namespace called `System.Reflection.Emit`

which is used in order to create a new assembly. This namespace contains builders for numerous types of constructs such as:

- AssemblyBuilder
- ModuleBuilder
- TypeBuilder
- MethodBuilder
- ConstructorBuilder

The builders are used in order to create the different parts of the assembly.

A class called `ILGenerator` is used to make the body of the methods. It contains methods for emitting the set of IL instructions and constructs directly related to IL code. In order to emit IL code, a class called `OpCodes` is providing all the different IL instructions for the `ILGenerator` to emit.

7.2.3 IL Reader

The class library provided by the .NET framework only allows users to load assemblies and extract information about the assemblies' content like e.g. which types and methods are available. Accessing the actual executable code is not provided by the .NET class library and it is therefore not possible to examine the CIL instructions within a method's body. A tool called IL Reader[1] is therefore used to extract the executable content of an assembly. IL Reader is a class library (IlReader.dll) which extends the existing `System.Reflection` API with functions for accessing the CIL code by reading the binary code of a CLI assembly file. This provides a fairly easy way to access CIL code from .NET assemblies which is not supported by the default reflection method.

The library contains a class called `ModuleReader` that provides easy access to a module. This includes getting the body of the method and getting referenced assemblies, modules and members.

Furthermore the IL Reader contains a class `MethodBody` to hold the method body. It also contains methods for getting information about the method. The information that is provided are the CIL instructions, the local variables and the exceptions a method can catch. Besides this, the code size and the maxstack provided.

In order to hold the information of an instruction, a wrapper class `Instruction` is provided. From this wrapper the CIL instruction, its offset and the operand can be read.

[1] http://www.aisto.com/roeder/dotnet

Chapter 8

The weaving process

The truth is rarely pure, and never simple.

— Oscar Wilde

This chapter describes different aspects of the implementation of the weaving process in the Aspect.NET compiler. The weaving process is the process in which the Aspect.NET compiler weaves the aspects into the CIL code of an existing assembly.

The weaving process, can be split into four stages. These four stages are the compiling of the aspects, the cloning of an assembly that should have aspects applied, the merging of the aspects into the assembly and the writing of the new assembly. On Figure 8.1 the four stages of the weaving process are illustrated.

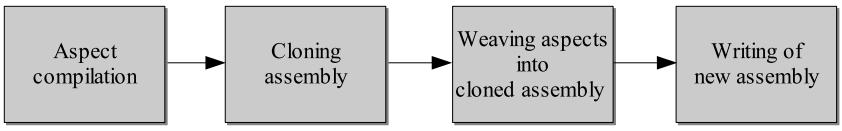


Figure 8.1: The four stages of the weaving process.

In the case where the aspects are implemented using inline code, the first stage starts by compiling the aspects into a DLL file. If the aspect implementation is provided in an extern DLL this stage can be skipped. The next stage is the making of an exact clone of the assembly into which the aspects are to be woven. The next stage is where the actual weaving of the aspects into the cloned assembly takes place. The last stage is where the cloned assembly together with weaved aspects is emitted and saved to a new file for execution. These stages will be described in more detail in the following sections.

8.1 Aspect compilation

The implementation of the aspects can be done in two different ways. One way is to reference a method in a .NET assembly which has already been compiled. The other way is to write the code inlined by enclosing it with the "`<? ?>`" tags and Aspect.NET will then create the .NET assembly which contains the implementation of crosscutting concerns.

If the implementation of crosscutting concerns is written by using tags within the aspect definition then the Aspect.NET compiler needs to extract the code between the tags and compile this separately by the corresponding language compiler. Figure 8.2 illustrates the process. Currently the Aspect.NET compiler supports both C# and F# for inline coding. If C# is used for inline coding the Aspect.NET compiler creates a new .cs file to which it inserts the required namespaces from the .dna source file, creates the specified namespace of the aspect and creates a class with the name of the aspect. To illustrate how inlined C# code is transfered to the separate .cs file consider the following aspect which intercepts console outputs:

```
using System;
namespace AspectNS
{
  aspect CSharp
  {
    before (string s) : call ( void Console.WriteLine(string) ) &&
                        args ( s )
    <?C#
      Console.WriteLine("Before "+s)
    ?>
  }
}
```

The .cs file which the Aspect.NET compiler generates for the above aspect will then look as follows:

```
using System;
namespace AspectNS
{
  public class CSharp
  {
    public static void CSAspectMethod0(System.String s)
    {
      Console.WriteLine("Before "+s);
    }
  }
}
```

The .cs file contains a class with the name of the aspect and the class is defined within the namespace of the specified aspect. The class has a static method with an auto-generated name which encapsulates the implementation of the before-advice. The .cs file is then compiled by the C# compiler (`csc.exe`) to a DLL

file which can be used in the same way as if the user provided a separate DLL file with the implementation of crosscutting concerns.

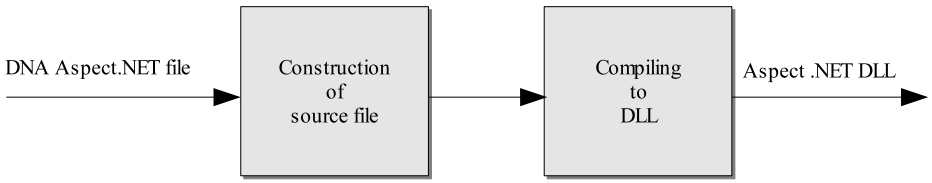


Figure 8.2: The aspect compiling process.

If the aspect instead uses F# for implementing crosscutting concerns the aspect would instead look like the following:

```
using System;
namespace AspectNS
{
  aspect FSharp
  {
    before (string s) : call ( void Console.WriteLine(string) ) &&
                        args ( s )
    <?F#
      Console.WriteLine("Before "^s)
    ?>
  }
}
```

The Aspect.NET compiler will in this case create a .ml file for compilation with the F# compiler. This .ml file looks as follows:

```
open System
let FSAspectMethod0 (s:string) = Console.WriteLine("Before "^s)
```

Since F# does not support namespaces and classes this information is not transferred to the .ml file and the aspect's name will therefore be lost once the aspect is compiled to CIL. The .ml file is compiled by the F# compiler (`fsc.exe`) to a DLL file, just as it was the case for C#.

If the aspect definition has made use of the primitive types supported by Aspect.NET these types must be transferred to their appropriate types in the generated .cs or .ml file. This can also be seen in the two previous examples where the Aspect.NET type `string` is converted to a `System.String` in C# and to a `string` in F#. Table 8.1 illustrates how the Aspect.NET compiler represents its primitive types in both F# and C#[1].

[1]Note that C# by default uses the .NET framework classes.

Basic data types		
Aspect.NET type	**FSharp type**	**.NET framework classes**
byte	FS.byte	System.Byte
sbyte	FS.sbyte	System.SByte
string	FS.string	System.String
short	FS.int16	System.Int16
int	FS.int32	System.Int32
long	FS.int64	System.Int64
ushort	FS.uint16	System.UInt16
uint	FS.uint32	System.UInt32
ulong	FS.uint64	System.UInt64
float	FS.float	System.Single
double	FS.float64	System.Double
char	FS.char	System.Char
bool	FS.bool	System.Boolean

Table 8.1: The Aspect.NET compiler's mapping between types in Aspect.NET, F# and the .NET framework classes.

8.2 Cloning assembly

Before the weaving process can begin it is necessary to make an exact clone of an assembly. In order to do this, the entire set of instructions together with methods, types, fields, etc. in the original managed assembly needs to be copied to a new dynamic assembly by using the `System.Reflection.AssemblyBuilder` library.

It is not a trivial task making an exact clone of an assembly, since the .NET class library does not support cloning of existing assemblies. The entire content of the source assembly must therefore be copied manually in order to make a copy. A variety of rarely used language constructs also complicates this cloning process. The hierarchy of the different constructs that should be copied is illustrated on Figure 8.3.

External references

The process of copying an assembly initiates by transferring the types and methods of the original assembly into the new assembly. The methods of the types are then created. This is done without emitting the body of the methods, hence all the methods needs to be created before the body of the methods is emitted - in order to make calling references to the new assembly. If the methods and its fields are not created first, external references to the original method in the original assembly, and not the new copy, will be made whenever a call to a method is emitted. This is illustrated on Figure 8.4. Here is a method `Foo()` calling a method `Bar()`. In the case where all the methods have not been created an external reference is made, meaning that the method `Bar()` in the original assembly is called instead of the method `Bar()` in the copied assembly.

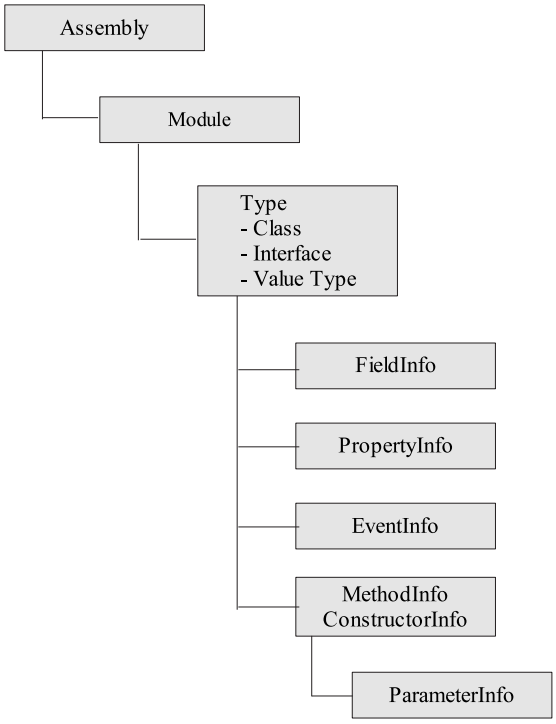


Figure 8.3: Assembly constructs hierarchy.

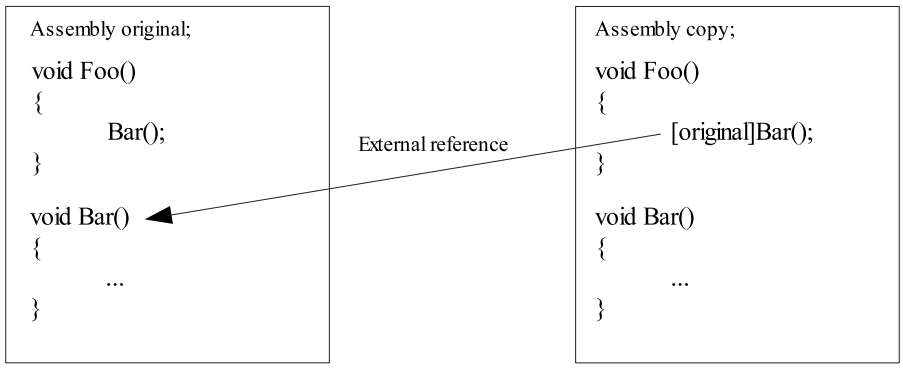


Figure 8.4: Clone of assembly with external reference.

The solution to this is to make all the empty methods (prototypes) and declare the local variables. The body of the method, together with all necessary information in order to emit the body, is saved in a method wrapper. The method wrapper contains lists with all the information needed to construct a method. All the CIL code, together with the local variable, branches, exceptions and a generator for the CIL code, is stored in the method wrapper.

The following sections describe the cloning of the method body which are the CIL instructions.

Branching

Branching instructions are control flow instructions used with e.g. if-structures. In CIL code, several types of branching instructions are represented. In section 2.5.2 the different types of branching instructions are described.

Each branching instruction consists of an instruction from section 2.5.2 and a label. The label represents the instruction that should be branched to. To fill in the label with an instruction to branch to, it needs to be marked in the CIL code where the branch branches to. Here is an example:

```
...
IL_0010 br  IL_005a
...
IL_005a ret
```

In the above example, a simple branching is illustrated. The `br` on address IL_0010 have a label that points to address IL_005a, which in this example is a `ret` (return instruction).

In order to make the label point to the instruction IL_005a, it is marked right before the emission of the return instruction. When the return statement is emitted, the label points to this instruction.

As long as the branch points to an instruction after the `br`-instruction, there is no problem as the instruction which should be marked is known when the instruction is reached. This could e.g. be an “if” structure.

Loop structures such as “for” and “while” however introduce a problem. A loop branch points to an instruction before the `br` instruction and this is the problem. For example is a while loop like this

```
...
while(i < 10){
...
}
...
```

converted into CIL code that looks like this

```
...
IL_0002 br.s     IL_001d
```

```
IL_0004 ...
...
...
IL_001d ldloc.0
IL_001e ldc.i4   10
IL_0020 blt.s    IL_0004
...
```

The first CIL instruction at IL_0002 is a branch to the comparison of a variable with `10`. The `blt` only branches back to IL_0004, if a variable is less than 10 and the loop runs one more time.

Since the instructions are emitted in a linear fashion, from start to end, all the branches are necessary to be known before the emission. This is to be able to mark the label at the emission of the relevant CIL instruction. This situation is also referred to as Backpatching[2].

So while the CIL code is read, a list of all the branches is created, so whenever a branching instruction needs to be "labeled" while it is emitted, the appropriate label is located in the list and marked.

Try-catch

The emission of a try-catch block introduces another problem. A try-catch block contains a block of code that is checked if an exception is thrown. If it is, the exception is caught and some optional code is executed.

In the following CIL code example, a method `testexc()` is tested if it throws an exception. If so, the exception is caught and the method `WriteLine()` is called writing the text "Catching an exception" to the screen.

```
try
{
  Test.testexc(true);
}
catch(System.Exception)
{
  Console.WriteLine(‘‘Catching an exception’’);
}
```

This try-catch block looks like the following when translated into CIL code:

```
.try
{
  IL_00ad:  ldc.i4.1
  IL_00ae:  call       void TestNS.Test::testexc(bool)
  IL_00b3:  leave.s    IL_00c3
}  // end .try
```

[2]For a definition of Backpatching see http://www.xp123.com/wwake/patterns/pat9906.shtml

```
catch [mscorlib]System.Exception
{
  IL_00b5:  stloc.s    V_6
  IL_00b7:  ldstr      "Catching an exception"
  IL_00bc:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_00c1:  leave.s    IL_00c3
}  // end handler
  IL_00c3 ret
```

A try-catch block is defined as two blocks of code, a try block and a catch block. When a try block is started, the CIL code that is emitted hereafter is contained inside the try block. When the catch block is started, the try block is automatically ended and when the CIL code that should be included inside the catch is emitted, the whole try-catch block is ended.

The problem arises under the copying of the try-catch block. Especially when the catch block is started and when it is ended. Whenever the catch block is started, it makes sure to end the try block. By doing this, an extra instruction, `leave`, is inserted at the end of the try block. The result of this is that the try block in the above example will look like this:

```
.try
{
  IL_00ad:  ldc.i4.1
  IL_00ae:  call       void TestNS.Test::testexc(bool)
  IL_00b3:  leave.s    IL_00c3
  IL_00c3:  leave      IL_005b
}  // end .try
```

The reason why this extra `leave` is inserted, is that when copying the body of the block each instruction is copied including the "old" `leave.s` instruction. As the methods for starting and ending the catch block insert a new `leave` instruction, the result are two `leave` instructions. To solve this, the last instruction in a try block and catch block is omitted in the emission of the body so only the `leave` instruction will be the auto-generated instruction.

Switch-case

A switch-case structure works the same way as normal branching instructions. The difference is that a switch can branch to one of several different instructions, depending on an integer value. The set of instructions that can be jumped to is called a jump-table.

An example of a switch-case structure is illustrated in the following:

```
switch(i)
{
  case 1:
    Console.WriteLine("1");
    break;
```

```
    case 2:
      Console.WriteLine("2");
      break;
    case 3:
      Console.WriteLine("3");
      break;
}
```

This switch-case looks like this when translated into CIL code:

```
IL_0007:  switch     (
                      IL_001a,
                      IL_0026,
                      IL_0032)
IL_0018:  br.s       IL_003e
IL_001a:  ldstr      "1"
IL_001f:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0024:  br.s       IL_003e
IL_0026:  ldstr      "2"
IL_002b:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0030:  br.s       IL_003e
IL_0032:  ldstr      "3"
IL_0037:  call       void [mscorlib]System.Console::WriteLine(string)
IL_003c:  br.s       IL_003e
IL_003e:  ret
```

Here is a switch instruction with a jump-table of three instructions. The instruction that should be jumped to, is decided by the current value on the stack. In this case if the value on the stack is 0, the first instruction IL_001a is jumped to. If the value is 1, it is the instruction IL_0026 that is jumped to, and if it is 2 the jump goes to IL_0032. If the value exceeds these values, a branch points to the end of the switch-case.

The emission of a switch-case is done much like a normal branching instruction, except that an array of labels is made and marked and emitted together with the switch instruction.

8.3 Weaving aspects into cloned assembly

When a clone of the assembly has been created, the code is ready to be scanned for method calls that should have aspects applied.

In order to apply the aspects, the CIL code is checked for method calls which have been defined as crosscutting concerns in the Aspect.NET source file. This is done by looking for the `call` and `callvirt` instructions described in section 2.5.2. Whenever one of these instructions is found, a check is made on whether or not it is a pointcut in the aspect definition and if aspects should be applied to it.

In the case where aspects should be applied, a check is made on what type of aspect that needs to be applied. If a before or after advice should be applied,

a method call is inserted before or after the pointcut respectively. If an around advice should be applied, the original method call is replaced with a new method call.

The difference between the way these advices are applied is that when the after and around advice are applied a check is made to see if a before advice has already been added. This is because the stack ordering needs to be saved. In the case where a before advice has been added, the stack ordering is already saved and there is no need to save the stack ordering again for the around and after advice. The problem of stack ordering is described in the next section.

8.3.1 Stack ordering for before and after advice

If a pointcut is intercepted, a check is made on the method to see if it needs parameters. If this is the case the parameters are on top of the stack when the call instruction is executed.

Figure 8.5: The parameter on top of the stack for method Point.SetX(int newX).

On Figure 8.5 the method `Point.SetX(int newX)` takes as parameter an integer. This means that the integer value that should be passed as parameter needs to be on top of the stack, when the method is called. If there are more parameters, the value on the top of stack is passed as the last parameter, the second value of the stack the next to last parameter and so forth.

This ordering of parameters on the stack introduces the problem when either a before or after advice is applied to a method. If the advice has parameters, they also need to be pushed onto the stack, before the advice call is made. The problem lies in the fact that the ordering of the parameters for the original method is no longer valid after the advice is executed since the top values on the stack are consumed by either the before or after advice.

On Figure 8.6 this problem is illustrated with the method `Point.SetX(int newX)` that has a before advice applied. The before advice `Advice.BeforeSetX(int X)` takes as parameter an integer and in this case the integer 5 is passed as parameter and therefore pushed onto the stack. Whenever the before advice is done executing, the original method `Point.SetX(int newX)` executes. This implies that an extra integer now is on top of the stack instead of the original integer. The problem is that the ordering and the values of the parameters on the stack differs from the stack ordering before the advice was applied, meaning that the wrong integer is

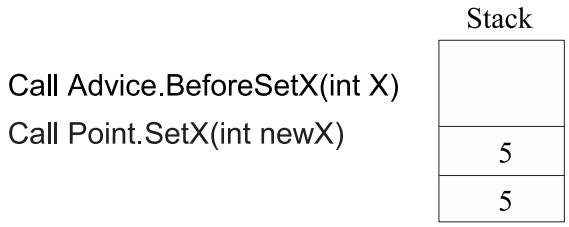


Figure 8.6: Before advice applied to Point.SetX(int newX).

passed as parameter to `Point.SetX(int newX)`. So in order to prevent stack underflow the original stack ordering needs to be restored.

In order to solve this, the original stack ordering is saved using additional local variables. This can be done by adding the instruction `stloc` for each parameter which saves the current value on top of the stack to a variable. By doing this, the original ordering is available at a later time and can be loaded by using the instruction `ldloc` including the parameter passed to the advice.

To illustrate this, the following CIL code shows the method `Point.SetX(int newX)` that is having its parameter from the result of the `Point.GetX()` method added with some argument:

```
...
IL_0001:  ldarg.0
IL_0002:  call       instance int32 Main.Point::GetX()
IL_0007:  ldarg.1
IL_0008:  add
IL_0009:  call       instance void Main.Point::SetX(int32)
...
```

If the before advice from Figure 8.6 is applied, it will look like this:

```
...
IL_0001:  ldarg.0
IL_0002:  call       instance int32 Main.Point::GetX()
IL_0007:  ldarg.1
IL_0008:  add
IL_0009:  stloc.s    V_1
IL_000b:  stloc.s    V_0
IL_000d:  ldloc.s    V_1
IL_000f:  call       void Advice.Advice::BeforeSetX(int32)
IL_0014:  ldloc.s    V_0
IL_0016:  ldloc.s    V_1
IL_0018:  call       instance void Main.Point::SetX(int32)
...
```

Here it can be seen that the stack ordering is saved using the `stloc` into the variables `V_0` and `V_1`, where after the value in variable `V_1` is pushed onto the

stack for the before advice. After the execution of the before advice, the original stack ordering is restored.

Call-by-reference

To handle parameters that are passed by call-by-reference, a check is made on the parameters whether they are references or not. If they are passed as references, the instruction `ldloca` (described in section 2.5.2) is inserted whenever the parameter should be pushed onto the stack. The difference between the instruction `ldloca` and the instruction `ldloc` is that `ldloca` loads the address for a value onto the stack, whereas `ldloc` loads the value onto the stack.

Implementing cflow

To control that the advices are only used in certain flows, cflow is implemented to check that the current control flow is correct. This check is implemented by using a stack for every control flow that is defined in the aspect definition. Every time a method is called which has a cflow, an array containing possible arguments and target is pushed onto the cflow's stack. Once the method returns the cflow's stack is popped. Whether the call occur in a certain context is implemented by checking if the stack is empty or not. If the stack is not empty the defined cflow is active and this information is used to determine whether the advice should be executed or skipped.

The following CIL code shows an example of how the Aspect.NET compiler implements a cflow which has been applied to the method `Graphics.Line.MoveBy(int,int)` and does not make use of `args` or `target`.

```
IL_001e: ldsfld   class NETAspect.CFlowStack
                  AspectNET.CFlowContainer::CFlow$Stack$Number$0
IL_0023: ldc.i4.0
IL_0024: newarr   [mscorlib]System.Object
IL_0029: callvirt instance void
                  [mscorlib]System.Collections.Stack::Push(object)
IL_002e: callvirt instance void Graphics.Line::MoveBy(int32,int32)
IL_0033: ldsfld   class NETAspect.CFlowStack
                  AspectNET.CFlowContainer::CFlow$Stack$Number$0
IL_0038: callvirt instance object
                  [mscorlib]System.Collections.Stack::Pop()
IL_003d: pop
```

The Aspect.NET compiler creates a separate class called CFlowContainer which has a field of type CFlowStack for every cflow in the aspect definition. Since the above example does not make use of `args` or `target` the array that is pushed onto the CFlowStack is of size 0. CFlowStack inherits from `System.Collections.Stack` which is why the methods `Push` and `Pop` are called on this class.

Now that cflow is active on the method `Graphics.Line.MoveBy(int,int)` and this information is available in the field `CFlow$Stack$Number$0` of the

class `CFlowContainer`. The following example shows how Aspect.NET checks whether the call to the method `Graphics.Point.MoveBy(int,int)` is performed in the context of the cflow that was implemented in the previous example.

```
IL_0000:  ldarg.0
IL_0001:  ldarg.1
IL_0002:  ldarg.2
IL_0003:  stloc.s    V_2
IL_0005:  stloc.s    V_1
IL_0007:  stloc.s    V_0
IL_0009:  ldsfld     class NETAspect.CFlowStack
                     AspectNET.CFlowContainer::CFlow$Stack$Number$0
IL_000e:  callvirt   instance bool NETAspect.CFlowStack::IsNotEmpty()
IL_0013:  brfalse    IL_0021
IL_0018:  ldloc.s    V_1
IL_001a:  ldloc.s    V_2
IL_001c:  call       void testaspect.myaspect::AspectMethod0(int32,
                                                             int32)
IL_0021:  ldloc.s    V_0
IL_0023:  ldloc.s    V_1
IL_0025:  ldloc.s    V_2
IL_0027:  call       instance void Graphics.Point::MoveBy(int32,
                                                          int32)
IL_002c:  ret
```

When a method with a cflow restriction is found its `CFlowStack` is loaded and the method `IsNotEmpty` is called to find out if the call is inside the flow. If the result is false, a branch makes a jump to the instruction line below the before call. If true the next line executed.

When an around-advice is used extra care has to be taken into extra consideration because normally the old call instruction is replaced by the around advices. In case of cflow, however, the old instruction has to be preserved such that it can still be executed if the flow is wrong. Therefore a branching instruction must be included in order to be able to call both methods.

8.4 Writing of new assembly

When the aspects have been applied to the new assembly, it is time to finish creating the methods and types, and write the new assembly to a file.

The first step in this process is to emit the code in the methods. This is done by running through each of the methods and emit the code for each method.

After the emission of the code for all the methods, all the types need to be finalized by calling the method `CreateType()` for each type. After this is done, the methods and the types are ready to be executed, and the last step is to write the new assembly to a file.

Part IV

Usage

This part describes how to use the Aspect.NET compiler and gives concrete examples of how Aspect.NET can be used to capture and modularize crosscutting concerns.

Chapter 9

Using Aspect.NET

This section will illustrate, mainly through examples how Aspect.NET is to be used. Section 9.1 explains how to use the Aspect.NET compiler such as which options are available when compiling and how the compiler outputs warnings and errors. Last section 9.2 gives code examples that illustrate the power of Aspect.NET.

9.1 The compiler

This section focuses on how to use the Aspect.NET compiler. The Aspect.NET compiler has the filename netaspect.exe and if entered into the command prompt, produces the following output (a fatal error occurs because no target file is specified):

```
Aspect.NET (C) Compiler version 0.0.4.1 for Microsoft
.NET Framework version 1.1 or newer.
Copyright (C) Bjoern Rasmussen, Casper S. Jensen,
Jimmy Nielsen and Lasse Jensen. All rights reserved.

fatal error: No inputs specified.
```

The compiler is depending on the two DLL files IlReader.dll and Tools.dll which must be located in the same directory as netaspect.exe. Note that when executing netaspect.exe it must reside in the same directory as the target assembly file. This is required in order to compensate for a bug in the assembly loader in the .NET framework.

The command "netaspect.exe /?" produces a help menu:

```
Aspect.NET (C) Compiler version 0.0.4.1 for Microsoft
.NET Framework version 1.1 or newer.
Copyright (C) Bjoern Rasmussen, Casper S. Jensen,
Jimmy Nielsen and Lasse Jensen. All rights reserved.
```

```
netaspect [options] <input file.dna> <target file>

       Aspect.NET Compiler Options

                        - OUTPUT FILES -
/out:<file>             Output a file name (default: uses base class,
                        so Main.exe --> NewMain.exe

                        - INPUT FILES -
/reference:<file>       Reference metadata from the specified
                        assembly file (short /r:)

                        - Errors and Warnings -
/doc                    Produces a text file (AspectCompileLog.log)
                        that logs all error messages that occured at
                        compile time
/doc:<file>             Produces a text file that logs all error
                        messages that occurred at compile time.

                        - MISCELLANEOUS -
/help                   Produces this user message (short /?)
/nologo                 Suppress compiler copyright message
```

As it can be seen from the Aspect.NET compiler help, the compiler includes some of the standard compiler options.

As an example of how to use the compiler, the below command is illustrated:

```
netaspect Trace.dna MainClass.exe
```

This command outputs the file NewMainClass.exe which is the .NET assembly that contains both the source program together with the woven aspects. If no specific output file is specified, the output file is named according to the filename of the source assembly with an extended New appended in front. Trace.dna is the source code that contains the actual aspects written in Aspect.NET, and the files containing the aspect code should always have the dna file extension, which is an abbreviation for Dot Net Aspect.

9.1.1 Dependencies

When the Aspect.NET compiler starts the weaving process the compiler requires that all assemblies, that are being referenced from the targeted assembly, are available in the search path. On Windows this will usually be through the environment variable PATH which contains locations of EXE and DLL files. The referenced assemblies must be available because the Aspect.NET compiler requires access to all the used types, methods, etc. when cloning the targeted assembly.

If users choose to write the implementation of crosscutting concerns within the aspect definition (the .dna file) then the compiler of the language which follows

the starting <? must be available to the Aspect.NET compiler. For C# this would require the csc.exe to be available in the PATH environment variable and for F# fsc.exe.

9.1.2 Error handling

Like any other compiler, Aspect.NET has a mechanism for handling errors. Below is illustrated how errors are handled in the Aspect.NET compiler:

```
Aspect.NET (C) Compiler version 1.0.4.1 for Microsoft
.NET Framework version 1.1 or newer.
Copyright (C) Bjoern Rasmussen, Casper S. Jensen,
Jimmy Nielsen and Lasse Jensen. All rights reserved.

Syntax error:
specific error message
```

Errors can be either syntactical or fatal. A fatal error occurs if e.g. no file (or the filename are misspelled) is specified as input. A syntactical error occurs if the syntax is not correct. As a help to the developer, the line number and character number are specified to ease the correction of syntax errors. Also if an additional error message exist, this is displayed, just below the standard error output.

Warnings are also generated, when a specified `call` do not evaluate to true, through the compilation. This means that if the method specified in the `call` is not used in the the assembly, the warning makes it possible to see if there is an error.

9.2 Code examples

This section gives concrete code examples to illustrate the power of Aspect.NET. Section 9.2.1 gives a Hello World example where advices are inserted into a hello world program. Section 9.2.2 gives an example, where a C# method is exchanged with an F# method. Section 9.2.3 gives an additional example where the synchronization of threads is in focus.

9.2.1 Hello World

This example illustrates how to make a simple “Hello World”-like program using Aspect.NET. The compiled program writes “Hello” to the console and after this waits for the user to input his name. The program then writes “Hello ” followed by the written name. Below is illustrated the source code for this program written in C#:

```
class Hello
{
        [STAThread]
```

```
        static void Main(string[] args)
        {
                SayHello();
                string name = System.Console.ReadLine();
                SayHelloTo(name);
                System.Console.ReadLine();
        }
        public static void SayHello()
        {
                System.Console.WriteLine("Hello");
        }
        public static void SayHelloTo(string name)
        {
                System.Console.WriteLine("Hello " + name);
        }
}
```

The output of executing this program will be:

```
Hello
```

Here the program will wait for an input value. If the string "Bob" is entered the program will continue:

```
Hello Bob
```

There is no need to write something, enter can just be pressed and the program will continue and will simply write hello again.

By using an aspect this simple program can be extended such that the user gets information about what he/she is supposed to do and also that the program should not accept an empty name. The following aspect can solve these two issues:

```
aspect HelloCorrect {
     pointcut WriteName() : call(void Hello.SayHello());
     pointcut GetName(string name) :
                          call(void Hello.SayHelloTo(string))
                                    && args(name);

     after() : WriteName()
     <?C#
           System.Console.WriteLine("Please write you name:");
     ?>

     before(ref string name) : GetName(name)
     <?C#
           while (name == "") {
                System.Console.WriteLine("I need your name to continue
                                          \n" + "so please write it:");
```

```
                name = System.Console.ReadLine();
            }
        ?>
}
```

An after advice is set to intercept `Hello.SayHello()`, which writes a line to tell the user what to do and a before advice is set to `Hello.SayHelloTo(string)` which keeps looping until a non-empty string is returned. Note that

Now the program outputs the following:

```
Hello
Please write your name:
```

If enter is pushed now the following message is written:

```
I need your name to continue
so please write it:
```

9.2.2 Exchanging code

This example illustrates the power of using cross-language aspect weaving and thereby using the best of each language and achieve the best performance and stability possible. The targeted assembly calculates the Fibonacci numbers in a standard recursive way. Formula 1 illustrates the formula used to calculate the Fibonacci numbers.

$$\begin{aligned} &Fib(0) = 0, \\ &Fib(1) = 1, \\ &Fib(n) = Fib(n-1) + Fib(n-2), \qquad \text{for } n \geq 2 \end{aligned} \tag{1}$$

Below is illustrated the C# source code which calculates the Fibonacci numbers and outputs them to the console:

```
class Fibonacci
{
        static void Main(string[] args)
        {
                Console.WriteLine("-----------
                                  Fibonacci number -----------");
                Console.WriteLine("Fibonacci "+Fib(57922));
                Console.ReadLine();
        }

        public static long Fib(long n)
        {
                if (n == 0 || n == 1)
                   return n;
                else
```

```
                    return Fib(n - 1) + Fib(n - 2);
        }
}
```

Calculating the Fibonacci numbers in this standard recursive manner may causes several complications, when e.g. the number `n` becomes sufficiently large the program may throw a `StackOverFlowException`. To solve the `StackOverFlowException` problem, a Fibonacci function has been made in F# using tail recursion, which means that the stack size remains constant. Below is illustrated the F# implementation of the tail recursive Fibonacci function:

```
let rec fib_aux ((n:int64), (next:int64), (result:int64)) =
  if (n = Int64.of_int32 0) then (result)
  else fib_aux(Int64.sub n (Int64.of_int32 1),
               Int64.add next result, next)

let fib n:int64 = fib_aux (n, Int64.of_int32 1, Int64.of_int32 0)
```

In order to weave the F# fibonnaci function into C#, Aspect.NET is needed. A pointcut is created catching every time a call is made to the C# Fibonacci method. An around advice then exchanges this method with the Fibonacci function in F#. Below is illustrated the source code for the dna file where the aspect are defined in:

```
using Fibonacci;

namespace fibreplace
{
        aspect Trace
        {
                pointcut Exchanging(long i) :
                        (call (long Fibonacci.Fib(long))) && args(i);

                around (long i) : Exchanging(i)
                [fib.dll, Fib.fib(i)]
        }
}
```

However, even though this newly woven assembly do not suffer from the `StackOverFlowException` performance is still an issue in many situations. Therefore performance was tested to see whether this new assembly using an F# implementation could match the performance of the original assembly. Table 9.1 illustrates the result of this performance test, note that REC is an abbreviation for recursion and TR is an abbreviation for tail recursion.

In order to check the accuracy of this test, the test was performed on the same PC under exactly, same conditions. The result was that the newly created assembly performed the calculation of the Fibonacci number where n=50 in 10.0144 ms, whereas the original assembly performed the calculation in 734656.384 ms which is quite a boost in performance.

Performance test, n=50		
	C#	**F#**
REC vs TR	734656.384 ms	10.0144 ms

Table 9.1: Performance test of Fibonacci example.

This performance test clearly shows some of the advantages of using cross-language aspect weaving. The fibonacci number can also be calculated using iteration which would enable C# to deliver better or similar performance but finding a corresponding iterative algorithm for a recursive algorithm may in some cases be very difficult.

9.2.3 Synchronization of threads

This section illustrates how Aspect.NET can be used to synchronize threads. The thread example is an instance of the classical producer/consumer problem by having a wood storage which is being filled by forest workers (producers) and emptied by a sawmill and two kinds of exporters. Part of the source code is illustrated below:

```
/// The wood storage
public class Storage
{
  private Stack UnitStorage = new Stack();
  public void Push(object ob)
  {
    UnitStorage.Push(ob);
  }
  public object Pop()
  {
    return UnitStorage.Pop();
  }
  public long Count()
  {
    return UnitStorage.Count;
  }
  public bool HasQuantity(int nCount)
  {
    return (Count() >= nCount);
  }
}

/// The wood producer
public class ForestWorker
{
  private int name;
  private const int UNITS_PRODUCED = 1;
  private const int PRODUCTION_TIME = 10;
  private Storage storage;
```

```
  public ForestWorker(Storage s, int name)
  {
    storage = s;
    this.name = name;
    thread = new Thread(new ThreadStart(Work));
    thread.Start();
  }
  public void Work(){..}
  public void Cut(){..}
  public void Return()
  {
    for(int i = 0; i < UNITS_PRODUCED; i++)
      storage.Push(new object());
  }
}

/// A wood consumer
public class SawMill
{
  private const int UNITS_CONSUMED = 4;
  private const int PREPARATION_TIME = 6;
  private Storage storage;
  public SawMill(Storage s)
  {
    storage = s;
    thread = new Thread(new ThreadStart(Work));
    thread.Start();
  }
  public void Work(){..}
  public void Buy()
  {
    do
    {
      if(storage.HasQuantity(UNITS_CONSUMED))
      {
        Thread.Sleep(10);
        for (int i = 0;i< UNITS_CONSUMED;i++)
        {
          storage.Pop();
          Thread.Sleep(10);
        }
        break;
      }
      Thread.Sleep(5);
    } while (true);
  }
  public void Process(){..}
}

/// A wood consumer
```

```
public class FireWood
{
  private const int UNITS_CONSUMED = 6;
  private const int PREPARATION_TIME = 12;
  //... remaining similar to SawMill class
}

/// A wood consumer
public class Exporter
{
  private const int UNITS_CONSUMED = 8;
  private const int PREPARATION_TIME = 18;
  //... remaining similar to SawMill class
}

/// Start the application
public class Begin
{
  public static void Main()
  {
    Storage sto = new Storage();
    ForestWorker[] worker = new ForestWorker[6];
    for (int i= 0; i < 6; i++)
      worker[i] = new ForestWorker(sto,i);

    FireWood fire = new FireWood(sto);
    SawMill saw = new SawMill(sto);
    Exporter exp = new Exporter(sto);
    Thread.Sleep(120000);
  }
}
```

The above example starts by creating a storage and six `ForestWorker` object which at a random intervals increments the stack in the `Storage` object. The `FireWood`, `SawMill` and `Exporter` objects are threads which at random intervals checks whether the number of required units are available in the storage. Since several threads may check the storage size at the same time they may try to consume an empty stack and thereby provoke a "stack is empty exception" causing the application to crash.

Usually the example above would be solved by using a shared lock between all consumers but this results in the same code being repeated which is hard to maintain e.g. when additional consumers are added. Aspects can instead be used provide synchronization for all consumers and the aspect below shows how:

```
namespace Synchronization
{
  aspect Sync
  {
    pointcut countCheck(int count, Storage storage) :
```

```
      call(bool Storage.HasQuantity(int))
      && args(count) && target(storage);

    pointcut finishBuy() :
      call(void FireWood.Buy()) ||
      call(void SawMill.Buy()) ||
      call(void Export.Buy());

    before (int count, Storage storage) : countCheck(count, storage)
    [Sync.dll,Synchronization.AspectThread.BeforeBuy(count,storage)]

    after() : finishBuy()
    [Sync.dll,Synchronization.AspectThread.AfterBuy()]
  }
}
```

The pointcut `countCheck` in the above aspect captures all consumers calls to `bool Storage.HasQuantity(int)` for which it has a before-advice that provides a shared lock in a separate DLL file. The reason a DLL file is used instead of in-lining with <?..? > is because Aspect.NET does not support static crosscutting which is required in order to provide a shared lock. The shared lock is therefore implemented in a class in the provided DLL file. The implementation of this crosscutting concern is shown below:

```
namespace Synchronization
{
  public class AspectThread
  {
    private static Mutex mutex = new Mutex();
    public static void BeforeBuy(int count, Storage storage)
    {
      mutex.WaitOne();
      if(storage.Count() < count)
      {
        mutex.ReleaseMutex();
      }
    }
    public static void AfterBuy()
    {
      mutex.ReleaseMutex();
    }
  }
}
```

The `Mutex` object is the shared lock for all consumers of the wood storage and the call to `mutex.WaitOne()` ensures that only one thread can check the number of units available at the storage at any time.

Part V

Results

This part summarizes the results that have been achieved during the design and development of Aspect.NET. Possible improvements for Aspect.NET are also discussed in future work.

Chapter 10

Conclusion

The conclusion is the place where you got tired of thinking.

— Martin H. Fischer

In this master thesis we have designed an aspect-oriented language and a compiler which allow developers to modularize crosscutting concerns in a wide range of .NET languages. The aspect-oriented language is called Aspect.NET because it in many ways resembles the AspectJ language. Aspect.NET supports the most widely used features in the AspectJ language for capturing crosscutting concerns, which are `call`, `cflow`, `target` and `args`. Once a crosscutting concern is captured, the user can then act upon it with a `before`-, `after`- and `around`-advice. The Aspect.NET language provides users with two ways of implementing a crosscutting concern (or advice), either by referring to a static method in a class in an assembly, or by writing the implementation within the aspect definition. Aspect.NET does not restrict the implementation of a crosscutting concern to a single language but instead allows users to use their preferred language. This ability can also be used to implement parts of a software system in a different language than the rest of the system. We have e.g. given an example of how recursive algorithms written in an imperative language can be replaced by tail-recursive algorithms from a functional language, thereby increasing performance and avoiding a possible stack-overflow.

Compared to existing .NET-based AOP technologies, Aspect.NET already has broader support for capturing crosscutting concerns in .NET languages than Weave.NET. We would also argue that Aspect.NET provides users with a much simpler way of applying aspects than is the case in Weave.NET which uses an XML-based approached. Aspect.NET does not yet support as many features as are provided by AspectC# but as its name indicates AspectC# is restricted to the C# language and performs weaving at source code level instead of the CIL level.

Aspect.NET ensures language independence by capturing crosscutting concerns in CIL to which all .NET languages compile. Users of Aspect.NET must therefore be familiar with the way that their language compiler translates a program to CIL. One way of doing this is to use disassembler tools for assemblies, like

Visual Studio's Object Browser or IL DASM from the .NET framework SDK, in order to locate the generalized-procedures which represent crosscutting concerns. Once a program has been compiled to CIL it may, however, not always be an easy task to map the compiled program to the program written in the high-level language. This poses a significant problem to users of Aspect.NET because they must be able to identify the method in CIL which represents a crosscutting concern.

The reason that some language compilers generate CIL code which is very difficult to analyze is that the high-level language has language constructs which are not supported by CIL and the compiler developer therefore has to use tricks in order for CIL to simulate the language construct. The design of CIL has to a great extent been based on supporting object-oriented programming languages and this, in some cases, makes translating languages from other paradigms very difficult. A program written in SML.NET which includes a high-order function is e.g. almost impossible to reason about once compiled to CIL. Steps are, however, taken towards extending the CIL such that more language constructs are supported. One example is the F# compiler which generates CIL code that has a very clear mapping from high-level language to CIL, even though it is a ML-language just like SML.NET. The reason for this is that F# makes use of a library which simplifies the use of functional language constructs resulting in less mangled CIL code.

Although some languages are quite hard to use with Aspect.NET, because of code mangling during translation to CIL, Aspect.NET is able to provide .NET developers with a simple way of using AOP in numerous languages and even mixing languages. Aspect.NET works particularly well with object-oriented languages like C#, J#, Visual Basic.NET, Managed C++, JScript, Component Pascal and to some extend also the functional language F#.

10.1 Future work

Although Aspect.NET includes a wide set of features for capturing and modularizing crosscutting concerns there are still many ways to improve Aspect.NET. The following sections outlines some of the features that will make the Aspect.NET language even more powerful and also ways to simplify the capture of crosscutting concerns in CIL code.

10.1.1 Wider support of AspectJ features

Since Aspect.NET has been developed with a basis primarily on AspectJ but still lacks many of the features available in AspectJ, it would naturally be desirable to support additional AspectJ features in Aspect.NET. Some of these features are e.g. the ability for aspects to inherit from other aspects and thereby simplify refinement. Additionally Aspect.NET only supports the standard `after`-advice whereas AspectJ has two special cases of this advice, namely `after returning` and `after throwing`. These two advices allow for users to examine return values and act upon exception, respectively. Currently Aspect.NET is only able to capture methods, while it may be desirable to be able to detect when an

attribute of a class or object is modified. This is e.g. supported in AspectJ by using the two pointcut designators `get` and `set`. Since CIL also supports global variables, which often cause problems that are hard to trace, these would also be desirable to support by the `set` and `get` pointcut designators.

There are, however, features in AspectJ which are debatable for whether they are desirable to include. One example is static crosscutting which can be used to extend classes by introducing new fields and methods. The use of static crosscutting, however, results in the entire description of a class being scattered across both Java and AspectJ source files, which makes it hard to get an overall picture of a class' properties and behavior.

10.1.2 Thread synchronization

Throughout this thesis a subset of the challenges associated with using multiple threads together with Aspect.NET have been presented. These challenges arises e.g. when a developer inserts an aspect around a method which is subject to thread synchronization. In this case the developer need to maintain the exact same synchronization in order to keep the application running as planned. Additionally since the aspects are inserted on the CIL-level, the developer might not even be aware of the fact that the aspects are inserted into a multi-threaded application.

Another challenge is when using multi-threading and cflow in Aspect.NET. For a multi-threaded application, a `CFlowStack` is needed for each thread, where in Aspect.NET only one `CFlowStack` is created regardless of the number of threads.

Creating these `CFlowStack`s need to be done at runtime because at compile time the number of needed threads is unknown. This is a rather complex task, since the creation and deletion of threads need to be located on the CIL-level. This involves an expansion of types because information like references to `CFLowStack`s have to be stored in the new types, instead of having it hardcoded in the CIL code.

Therefore improving the design to solve the challenges of thread synchronization and multi-threaded applications would be a significant improvement.

10.1.3 Cloning the assemblies

In the process of cloning the assembly, several features have been compromised. As many special language constructs needs special instructions and methods in order to clone them, a decision only to clone the basic language constructs was made. This implies that the cloning of more advanced constructs could be implemented in future releases.

Such constructs could e.g. be the cloning of global methods. Global methods are methods that does not belong to a class, and is thereby usable globally. Global methods are used in languages like C/C++ and Pascal. Events is another construct which is not yet supported by Aspect.NET's cloning of assemblies. Events are e.g. used by C# for event handling like when a button is pressed.

Currently the Aspect.NET compiler does not transfer debugging information

during the cloning of an assembly which can greatly complicate the search for errors since an application crash will no longer reveal where in the source code the error occured. If the assembly containing the implementation of crosscutting concerns also contains debugging information this information should be included using the weaving process.

10.1.4 Additional support for CIL constructs

Aspect.NET does not yet support all language contructs which are available in CIL and is therefore able to capture all crosscutting concerns. One example is e.g. when pointers are used as arguments or return types which is not yet supported by Aspect.NET and therefore cannot be used in a join point.

Another language construct which could be improved in Aspect.NET is the support of properties in CIL, which at compile time is changed to two methods, a get and a set method. Currently Aspect.NET is able weave aspect that will intercept reads and writes to properties, but this requires that the user uses a tool like IL DASM in order to find the corresponding method names which control the property. A better way to support this would be to let Aspect.NET locate the methods needed.

10.1.5 Graphical user interface

Throughout the development of Aspect.NET a tradeoff between flexibility and simplicity has been made, where flexibility often has had the highest priority. This also implies that the user-friendliness of the compiler has had a low priority compared to the functionality. Currently, in order to use Aspect.NET, the user needs to know how to use a disassembler tool, like IL DASM and also needs to have considerable knowledge of CIL code in order to insert aspects. One way of making Aspect.NET easier to use, is by developing a graphical user interface (GUI) for Aspect.NET. This GUI should then support the insertion of aspects. This means that the GUI should offer the same functionality like IL DASM where the user then should look at the disassembled assembly and through a point and click system insert the advices either before or after a method call. A GUI should only supplement the Aspect.NET compiler, so it is still possible to use the tools of choice. By offering a GUI this lowers the technical requirements of the developer and thereby reach a broader audience.

10.2 Reflections on AOP

Although AOP has shown to be useful in areas where traditional languages have weaknesses, it is, however, doubtful that AOP will become a programming paradigm in the same way as e.g. object-oriented programming. The reason for this gloom is that for a new programming style to be successful it, first of all, has to be straight forward and easy to use. This is hardly the case with AOP because users must really be aware of when aspects are in use and when they are not. The fact that aspects are separate from the actual “core functionality” of a program makes it hard to read what the program is actually doing.

Still AOP has significant advantages over traditional languages, particularly in applications where crosscutting concerns change frequently. Using traditional languages in such cases will simply be too time-consuming. Tool-support for aspect weaving may also simplify the use of AOP and thereby become applicable to a wider range of users. Borland's JBuilder[1] is an example of a tool which supports AOP by providing a plug-in for AspectJ such that users can browse the structure of aspects and view the crosscutting structure in their programs.

[1] http://www.borland.com/jbuilder/

Part VI

Literature

Chapter 11

Bibliography

[1] Mehmet Aksit. Composition and Separation of Concerns in the Object-Oriented Model. ACM, December 1996.

[2] Andrew Kennedy, Claudio Russo, Nick Benton. SML .NET 1.1 User Guide. Microsoft Research, November 2003.

[3] The AspectJ Team. AspectJ Programming Guide. Xerox Corporation, 2002.

[4] Bjarne Stroustrup. The C++ Programming Language. Addison Wesley, 2003.

[5] Brian W. Kernighan, Dennis Ritchie. C Programming Language (2nd Edition). Prentice Hall PTR, 1988.

[6] Charles Simonyi. Intentional Programming: Asymptotic Fun? Microsoft, 2004.

[7] Malcolm K. Crowe. Compiler Writing Tools using C#. University of Paisly, Scotland, January 2004.

[8] ECMA. Common Language Infrastructure (CLI) - Partition I: Concepts and Architecture. ECMA, October 2002.

[9] ECMA. Common Language Infrastructure (CLI) - Partition II: Metadata Definition and Semantics. ECMA, October 2002.

[10] ECMA. Common Language Infrastructure (CLI) - Partition III: CIL Instruction Set. ECMA, October 2002.

[11] ECMA. Common Language Infrastructure (CLI) - Partition IV: Profiles and Libraries. ECMA, October 2002.

[12] ECMA. Common Language Infrastructure (CLI) - Partition V: Annexes. ECMA, October 2002.

[13] Andreas Frei, Patrick Grawehr and Gustavo Alonso. A Dynamic AOP-Engine for .NET. Swiss Federal Institute of Technology Zurich, 2004.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. Springer-Verlag, 1997.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ.

[16] Howard Kim. Aspectc#: An AOSD Implementation for C#. Trinity College Dublin, 2002.

[17] Donal Lafferty and Vinny Cahill. Language-Independent Aspect-Oriented Programming. Trinity College Dublin, 2003.

[18] Serge Lidin. Inside Microsoft .NET IL Assembler. Microsoft Corporation, 2002.

[19] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. 1993.

[20] Erik Meijer, Jim Miller. Technical Overview of the Common Language Runtime (or why the JVM is not my favorite exection platform), June 2001.

[21] Microsoft Corporation. C# Language Specification 1.2, 2003.

[22] Microsoft Corporation. Visual Basic .NET Language Specification (seen May 23 2004). http://msdn.microsoft.com/library/en-us/vbls7/html/vbSpecStart.asp, November 2003.

[23] Microsoft Corporation. Active Template Library (ATL) Reference (seen June 6 2004). http://msdn.microsoft.com/library/en-us/vcmfc98/html/atl.asp, 2004.

[24] Microsoft Corporation. Component Object Model (seen June 6 2004). http://msdn.microsoft.com/library/en-us/dnanchor/html/componentobjectmodelanchor.asp, 2004.

[25] Microsoft Corporation. F# (seen May 23 2004). http://research.microsoft.com/projects/ilx/fsharp.aspx, 2004.

[26] Microsoft Corporation. JScript .NET (seen May 23 2004). http://msdn.microsoft.com/library/en-us/jscript7/html/jsmscStartPage.asp, 2004.

[27] Microsoft Corporation. Managed Extensions for C++ Specification (seen May 23 2004). http://msdn.microsoft.com/library/en-us/vcmxspec/html/vcmanagedextensionsspec_start.asp, 2004.

[28] Microsoft Corporation. Microsoft Foundation Class Library (seen June 6 2004). http://msdn.microsoft.com/library/en-us/vcmfc98/html/mfchm.asp, 2004.

[29] Microsoft Corporation. Visual JSharp (seen May 23 2004). http://msdn.microsoft.com/library/en-us/dv_vjsharp/html/vjgrfvisualjlanguageoverview.asp, 2004.

[30] Niklass Påhlsson. Aspect Oriented Programming. University of Kalmar, November 2002.

[31] Matt Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. MSDN, March 1994.

[32] Garden Point. Getting started with Gardens Point Component Pascal. Garden Point, September 2002.

[33] Raphael Simon, Emmanuel Stapf. Full Eiffel on the .NET Framework (seen May 23 2004). http://msdn.microsoft.com/library/en-us/dndotnet/html/pdc_eiffel.asp, July 2002.

[34] Don Syme. ILX: Extending the .NET Common IL for Functional Language Interoperability. Microsoft Research, Cambridge, U.K., 2001.

[35] Frank Yellin Tim Lindholm. The Java Virtual Machine Specification. Sun Microsystems, Inc., 1999.

Part VII

Appendix

This part contains appendices, which can be read by the interested reader and gives a deeper understanding of selected topics from the report.

Appendix A

BNF syntax for Aspect.NET

This appendix contains a complete Backus Naur Form (BNF) syntax for the Aspect.NET language. The symbols used are:

```
::=               Meaning "is defined as"
|                 Meaning "or"
<>                Angle brackets used to surround category names
```

Some predefined categories are defined in a special way. These are:

```
<id>              id is a text string used for naming. The string is
                  restricted by following regular expresion
                  [a-zA-Z_$@?][a-zA-Z0-9_$@?]*
<fsharpcodechar>  Each char in the string incapsulated by <?F# ?>
<csharcodechar>   Each char in the string incapsulated by <?C# ?>
```

The syntax are:

```
<documentstart>    ::= <Usings> <documents> | <documents>
<Usings>           ::= 'using' <namepath> ';' | 'using' <namepath> ';' <Usings>
<documents>        ::= <document> | <document> <documents>
<document>         ::= <aspects> | 'namespace' <namepath> '{' <documents> '}'
<aspects>          ::= <aspect> | <aspect> <aspects>
<aspect>           ::= <access_specifier> 'aspect' <id> '{' <aspectcodes> '}'
<namepath>         ::= <id> | <id> '.' <namepath>
<aspectcodes>      ::= <aspectcode> | <aspectcode> <aspectcodes>
<aspectcode>       ::= <pointcut> | <before> | <after> | <around>
<pointcut>         ::= <access_specifier> 'pointcut' <id> '(' ')' ':'
                       <pointdefs> ';' | <access_specifier> 'pointcut'
                       <id> '(' <defparameters> ')' ':' <pointdefs> ';'
<before>           ::= <access_specifier> 'before' '(' ')' ':' <modifycode> |
                       <access_specifier> 'before' '(' <defparameters>
```

```
                    ')' ':' <modifycode>
<after>           ::= <access_specifier> 'after' '(' ')'
                    ':' <modifycode> | <access_specifier> 'after'
                    '(' <defparameters> ')' ':' <modifycode>
<around>          ::= <access_specifier> 'around' '(' ')' ':'
                    <modifycode> | <access_specifier> 'around'
                    '(' <defparameters> ')' ':' <modifycode>
<pointdefs>       ::= <pointdef> | '(' <pointdefs> ')' | <pointdefs>
                    '&&' <pointdefs> | <pointdefs> '||' <pointdefs>
<pointdef>        ::= 'call' '(' <callmethod> ')' | 'args' '('
                    <parameters> ')' | 'target' '(' parameter ')' |
                    'cflow' '(' <pointdefs> ')' | '!' 'cflow' '('
                    <pointdefs> ')'
<modifycode>      ::= <namepath> '(' ')' <filemethodref> | <namepath>
                    '(' parameters ')' <filemethodref>
                    | <pointdefs> <filemethodref>
<parameters>      ::= <paramter> | <parameter> ',' <parameters>
<defparameters>   ::= <type> <id> | <type> <id> ',' <defparameters>
<parameter>       ::= 'ref' <id> | <id>
<parametertypes>  ::= <type> | <type> ',' <parametertypes>
<callmethod>      ::= <type> <namepath> '(' ')' | <type> <namepath>
                    '(' <parametertypes> ')' | <namepath> '.' '.'
                     <id> '(' ')' | <namepath> '.' '.' <id> '('
                     <parametertypes> ')'
<filemethodref>   ::= <programmingcode> | '[' <file> ',' method ']'
<programmingcode> ::= <csharpcodepieces> | <fsharpcodepieces>
<csharpcodepieces> ::= <csharpcodechar> | <csharpcodepieces> <csharpcodepieces>
<fsharpcodepieces> ::= <fsharpcodechar> | <fsharpcodepieces> <fsharpcodepieces>
<file>            ::= <id> '.' <id> | <id> ':' '\' <filelib> <id> '.' <id>
<method>          ::= <namepath> '(' ')'| <namepath> '(' <parameters> ')'
<type>            ::= <type> '[' ']' | <type> '[' <commainsert> ']' |
                    'ref' <namepath> | <namepath>
<commainsert>     ::= ',' | ',' <commainsert>
<filelib>         ::= <id> '\' | <id> '\' <filelib>
<access_specifier> ::= 'public' | 'private' | 'protected' | 'internal'
```

Appendix B

Lexer tokens

```
%lexer

%token ID {
        public string name;
}

%token ACCESSSPE {
        public string spe;
}
```

"private"|"public"|"protedted"|"internal"	%ACCESSSPE {spe = yytext;}
"using"	%USING
"namespace"	%NAMESPACE
"aspect"	%ASPECT
"pointcut"	%POINTCUT
"after"	%AFTER
"before"	%BEFORE
"around"	%AROUND
"call"	%CALL
"target"	%TARGET
"args"	%ARGS
"cflow"	%CFLOW
"ref"	%REF
":"	%COLON
";"	%SEMICOLON
"||"	%OR
"&&"	%AND
"!"	%NOT
"("	%LPAREN
")"	%RPAREN
"{"	%LBRAKE
"}"	%RBRAKE
"["	%LSQUARE
"]"	%RSQUARE

```
"."                                     %DOT
","                                     %COMMA
"\\"                                    %BACKSLACH
[a-zA-Z_$@?][a-zA-Z0-9_$@?]*            %ID {name = yytext;}
"<?C#"                                  {yybegin("CSHARPCODE");}
<CSHARPCODE>.                           %CSHARPCODECHAR
<CSHARPCODE>\n                          %CSHARPCODECHAR
<CSHARPCODE>"?>"                        {yybegin("YYINITIAL");}
"<?F#"                                  {yybegin("FSHARPCODE");}
<FSHARPCODE>\n                          %FSHARPCODECHAR
<FSHARPCODE>.                           %FSHARPCODECHAR
<FSHARPCODE>"?>"                        {yybegin("YYINITIAL");}
"/""/"                                  {yybegin("LINECOMMENT");}
<LINECOMMENT>.                          ;
<LINECOMMENT>\n                         {yybegin("YYINITIAL");}
"/""*"                                  {yybegin("COMMENT");}
<COMMENT>"*""/"                         {yybegin("YYINITIAL");}
<COMMENT>.                              ;
<COMMENT>\n                             ;
[\t\r\n ]                               ;
```

Appendix C

Parser syntax

```
%parser aspectNet.lexer

%left SEMICOLON BACKSLACH ACCESSSPE ASPECT ID AND OR
%before AND
%after OR

documentstart          : usings:u documents:d              %DocumentStart(u,d);
documentstart          : documents:d                       %DocumentStart(d);

usings                 : usin:u SEMICOLON                  %Usings(u);
usings                 : usin:u SEMICOLON usings:us        %Usings(u,us);

usin                   : USING namepath:n                  %Usin(n);

documents              : document:d                        %Documents(d);
documents              : document:d documents:ds           %Documents(d,ds);

document               : aspects:a                         %Document(a);
document               : namespac:n                        %Document(n);

namespac               : NAMESPACE namepath:n
                         LBRAKE documents:d RBRAKE         %Namespac(n,d);

aspects                : aspect:a                          %Aspects(a);
aspects                : aspect:a aspects:as               %Aspects(a,as);

aspect                 : aspectkey:a name:n LBRAKE
                         aspectcodes:acs RBRAKE            %Aspect(a,n,acs);

namepath               : name:n                            %Namepath(n);
namepath               : name:n DOT namepath:np            %Namepath(n,np);

name                   : ID:i                              %Name(i);
```

```
aspectcodes             : aspectcode:ac                          %AspectCodes(ac);
aspectcodes             : aspectcode:ac aspectcodes:acs          %AspectCodes(ac,acs);

aspectcode              : pointcut:p                             %Point(p);
aspectcode              : before:b                               %Befo(b);
aspectcode              : after:a                                %Aft(a);
aspectcode              : around:a                               %Aro(a);

pointcut                : pointcutkey:p name:n LPAREN
                          RPAREN COLON pointdefs:po SEMICOLON    %PointCut(p,n,po);
pointcut                : pointcutkey:p name:n LPAREN
                          defparameters:d RPAREN COLON
                          pointdefs:po SEMICOLON                 %PointCut(p,n,po,d);

before                  : beforekey:b LPAREN RPAREN
                          COLON modifycode:m                     %BeforeMod(b,m);
before                  : beforekey:b LPAREN defparameters:d
                          RPAREN COLON modifycode:m              %BeforeMod(b,m,d);

after                   : afterkey:a LPAREN RPAREN COLON
                          modifycode:m                           %AfterMod(a,m);
after                   : afterkey:a LPAREN defparameters:d
                          RPAREN COLON modifycode:m              %AfterMod(a,m,d);

around                  : aroundkey:a LPAREN RPAREN
                          COLON modifycode:m                     %AroundMod(a,m);
around                  : aroundkey:a LPAREN defparameters:d
                          RPAREN COLON modifycode:m              %AroundMod(a,m,d);

pointdefs               : pointdef:p                             %PointDefs(p);
pointdefs               : LPAREN pointdefs:ps RPAREN             %PointDefsPar(ps);
pointdefs               : pointdefs:p AND pointdefs:ps           %PointDefsAnd(p,ps);
pointdefs               : pointdefs:p OR pointdefs:ps            %PointDefsOr(p,ps);

pointdef                : CALL LPAREN callmethod:c RPAREN        %Call(c);
pointdef                : ARGS LPAREN parameters:p RPAREN        %Args(p);
pointdef                : TARGET LPAREN parameter:p RPAREN       %Target(p);
pointdef                : CFLOW LPAREN pointdefs:p RPAREN        %CFlow(p);
pointdef                : NOT CFLOW LPAREN pointdefs:p RPAREN    %NotCFlow(p);

modifycode              : namepath:n LPAREN RPAREN
                          filemethodref:f                        %ModifyPointCut(n,f);
modifycode              : namepath:n LPAREN
                          parameters:p RPAREN
                          filemethodref:f              %ModifyPointCutPar(n,p,f);
modifycode              : pointdefs:p filemethodref:f  %ModifyPointDefs(p,f);

pointcutkey             : POINTCUT                               %PointCutKey();
pointcutkey             : ACCESSSPE:a POINTCUT                   %PointCutKeyAcc(a);
```

```
beforekey              : BEFORE                                    %BeforeKey();
beforekey              : ACCESSSPE:a BEFORE                        %BeforeKeyAcc(a);

afterkey               : AFTER                                     %AfterKey();
afterkey               : ACCESSSPE:a AFTER                         %AfterKeyAcc(a);

aroundkey              : AROUND                                    %AroundKey();
aroundkey              : ACCESSSPE:a AFTER                         %AroundKeyAcc(a);

aspectkey              : ASPECT                                    %AspectKey();
aspectkey              : ACCESSSPE:a ASPECT                        %AspectKeyAcc(a);

parameters             : parameter:p                               %Parameters(p);
parameters             : parameter:p COMMA parameters:ps           %Parameters(p,ps);

callmethod             : type:t namepath:n LPAREN RPAREN           %CallMethod (t,n);
callmethod             : type:t namepath:n LPAREN
                         parametertypes:p RPAREN                   %CallMethodPar(t,n,p);
callmethod             : namepath:n DOT DOT name:n2
                         LPAREN RPAREN                             %Constructor(n,n2);
callmethod             : namepath:n DOT DOT name:n2
                         LPAREN parametertypes:p RPAREN            %ConstructorPar(n,n2,p);

parameter              : REF name:n                                %RefPar(n);
parameter              : name:n                                    %Parameter(n);

filemethodref          : programingcode:c                          %FileMethodRefCode(c);
filemethodref          : LSQUARE file:f COMMA method:m RSQUARE     %FileMethodRef(f,m);

programingcode         : csharpcodepieces:cs                       %CSharpCode(cs);
programingcode         : fsharpcodepieces:fs                       %FSharpCode(fs);

csharpcodepieces       : CSHARPCODECHAR:cc                         %CSharpCodeChar(cc);
csharpcodepieces       : csharpcodepieces:p1
                         csharpcodepieces:p2                       %CSharpCodeString(p1,p2);

fsharpcodepieces       : FSHARPCODECHAR:fc                         %FSharpCodeChar(fc);
fsharpcodepieces       : fsharpcodepieces:p1
                         fsharpcodepieces:p2                       %FSharpCodeString(p1,p2);

defparameters          : defparameter:d                            %DefParameters(d);
defparameters          : defparameter:dp COMMA
                         defparameters:dps                         %DefParameters(dp,dps);

file                   : name:n1 DOT name:n2                       %File(n1,n2);
file                   : filepath:f name:n1 DOT name:n2            %FilePath (n1,n2,f);

method                 : namepath:n LPAREN RPAREN                 %Method(n);
method                 : namepath:n LPAREN parameters:p RPAREN%MethodPar(n,p);
```

```
parametertypes          : type:t                                %ParameterTypes(t);
parametertypes          : type:t COMMA parametertypes:p         %ParameterTypes(t,p);

type                    : type:t LSQUARE RSQUARE                %TypeReaderArray(t);
type                    : type:t LSQUARE commainsert:c
                          RSQUARE                       %TypeReaderCommaArray(t,c);
type                    : REF namepath:n                        %TypeReaderRef(n);
type                    : namepath:n                            %TypeReader(n);

commainsert             : COMMA                                 %CommaInsert();
commainsert             : COMMA commainsert:c                   %CommaInserts(c);

defparameter            : type:t name:n                         %DefParameter(t,n);

filepath                : name:n COLON BACKSLACH filelib:f      %Filepath (n,f);

filelib                 : name:n BACKSLACH                      %FileLib(n);
filelib                 : name:n BACKSLACH filelib:f            %FileLib(n,f);
```

Appendix D

List of .NET languages

Table D.1 illustrates a list of all NET languages[1]. Note that the cells stating *Unknown**** is due to the fact that no manufacturer could be located. Also note that this list might not be complete, since new languages arise all the time, and it has not been possible to find someone who claims to have a complete list.

[1]This list was updated April 24^th and was inspired by the list on: http://weblogs.asp.net/britchie/archive/2003/03/17/3920.aspx

.NET compilers		
Language	**Project**	**Manufacturer**
ADA	A# - port of Ada to .NET	Dr. Martin C. Carlisle
APL	Dyalog.NET - Dyalog APL	Dyadic
C#	C#	Microsoft
	mcs	Mono/Ximian
	cscc	DotGNU Portable.NET
Caml	F# (ML and Caml), Abstract IL, ILX	Microsoft Research
C++	Managed Extensions for C++	Microsoft
	Managed and Unmanaged C++	GotDotNet
Cobol	NETCOBOL - COBOL for .NET	Fujitsu
	Net Express	Micro Focus
Delphi	Borland Delphi and C++ Builder Support for .NET	Borland
	Delphi.NET - interoperatibility tools	Marcus Schmidt
Eiffel	Eiffel for .NET	Interactive Software Engineering
Forth	Delta Forth .NET	Valer Bocan
Fortran	Lahey/Fujitsu Fortran for .NET	Lahey Computer Systems, Inc.
	FTN95 - Fortran for Microsoft .NET	Salford Software Ltd.
Java	Visual J# .NET	Microsoft
	Java VM for .NET	IKVM.NET
JavaScript	JSscript .NET	GotDotNet
	JANET - Javascript-compatible language	Steve Newman
LOGO	MonoLOGO	Richard Hestilow
Lua	Lua.NET: Integrating Lua with Rotor	PUC-RIO
Mercury	Mercury on .NET	The Mercury Project
Mondrian	Mondrian and Haskell for .NET	Nigel Perry
Oberon	Active Oberon for .NET	ETH Zuerich
Perl	Perl for .NET, PERLNET	ActiveState SRL
PHP	PHP Sharp	AK BK Home
Python	KOBRA	unknown***
	Open Source Python for .NET	Mark Hammond
Ruby	NetRuby	unknown***
RPG	ASNA Visual RPG for .NET	unknown***
Scheme	Scheme	Northwestern University
Small Talk	S#	SmallScript LLC
SML	SML.NET	Microsoft Research
Visual Basic	VB.NET	Microsoft
	mbas	Mono/Ximian

Table D.1: .NET languages.

Appendix E

Figure editor in C#

This appendix contains a C# version of the classical figure editor (from the AspectJ tutorial [3]) used to illustrate crosscutting concerns:

E.1 Interface: FigureElement

```
namespace Graphics
{
  /// <summary>
  /// Summary description for FigureElement.
  /// </summary>
  public interface FigureElement
  {
    void MoveBy(int dx, int dy);
  }
}
```

E.2 Class: Point

```
namespace Graphics
{
  /// <summary>
  /// Summary description for Point.
  /// </summary>
  public class Point : FigureElement
  {
    private int x = 0, y = 0;

    public Point(int x, int y): base()
    {
      SetX(x);
      SetY(y);
```

```
    }

    public int GetX() { return x; }
    public int GetY() { return y; }

    public void SetX(int x) { this.x = x; }
    public void SetY(int y) { this.y = y; }

    public void MoveBy(int dx, int dy)
    {
      SetX(GetX() + dx);
      SetY(GetY() + dy);
    }
  }
}
```

E.3 Class: Line

```
using System;

namespace Graphics
{
  /// <summary>
  /// Summary description for Line.
  /// </summary>
  public class Line : FigureElement
  {
    private Point p1, p2;

    public Line(Point p1, Point p2): base()
    {
      this.p1 = p1;
      this.p2 = p2;
    }

    public Point GetP1() { return p1; }
    public Point GetP2() { return p2; }

    public void SetP1(Point p1) { this.p1 = p1; }
    public void SetP2(Point p2) { this.p2 = p2; }

    public void MoveBy(int dx, int dy)
    {
      GetP1().MoveBy(dx, dy);
      GetP2().MoveBy(dx, dy);
    }
  }
}
```

E.4 Class: Display

```
using System;
using System.Collections;


namespace Graphics
{
  /// <summary>
  /// Summary description for Display.
  /// </summary>
  public class Display
  {
    static ArrayList objects = new ArrayList();

    public static void AddObject(Line line)
    {
      objects.Add(line);
    }

    public static void Update()
    {
      Console.WriteLine("Updating screen");
      for(int i=0;i<objects.Count;i++)
      {
        Line l = (Line)objects[i];
        Console.WriteLine("Line "+(i+1)+" coordinates ((x,y),(x,y)): (("+l.GetP1().GetX()+","+l
      }
    }

    /// the horizontal size of the display widht
    public static int GetWidth()
    {
      return 400;
    }

    /// the vertical size of the display width
    public static int GetHeight()
    {
      return 400;
    }
  }
}
```

E.5 Class: Figure

```
using System;
using System.Collections;
```

```
namespace Graphics
{
  /// <summary>
  /// Summary description for Figure.
  /// </summary>
  public class Figure
  {
    private ArrayList elements = new ArrayList();

    public Figure(){}

    public Point MakePoint(int x, int y)
    {
      Point p = new Point(x, y);
      elements.Add(p);
      return p;
    }

    public Line MakeLine(Point p1, Point p2)
    {
      Line l = new Line(p1, p2);
      elements.Add(l);
      Display.AddObject(l);
      return l;
    }
  }
}
```

E.6 Class: MainClass

```
using System;

namespace Graphics
{
  /// <summary>
  /// Summary description for Main.
  /// </summary>
  public class MainClass
  {
    public static void Main(string[] args)
    {
      Figure fig = new Figure();
      Line line = fig.MakeLine( new Point(0,0),
                                new Point(Display.GetWidth(),
                                          Display.GetHeight()));
      line.MoveBy(-50,-50);
      System.Console.ReadLine();
```

```
    }
  }
}
```