Twente Research and Education on Software Engineering,
Department of Computer Science,
Faculty of Electrical Engineering, Mathematics and Computer Science,
University of Twente

Performing transformations on .NET Intermediate Language code

S.R. Boschman

Enschede, August 23, 2006

Prof. dr. ir. M. Aksit
Dr. ir. L.M.J. Bergmans
Ir. P.E.A. Dürr

Abstract

To express crosscutting concerns in a clear manner, the aspect-oriented programming (AOP) paradigm was introduced. In AOP languages, crosscutting concerns are defined in *aspects*. These aspects are composed, or woven, with *components*. These components encapsulate functionality expressed in traditional object-oriented languages.

As the aspect language and component language can be separated, an AOP solution can be implemented independent of the component language. A suitable platform for such an AOP solution is the .NET Framework, since, in principle, this platform can support a wide range of different component languages (i.e. programming languages).

This thesis will describe the development of a tool that can weave the aspects with the components in the .NET Framework environment. The first step is to discuss the Common Intermediate Language (CIL) in more detail, as all programming languages in the .NET Framework map to this CIL. The second step is to create a mapping from AOP constructs to CIL code structures. The third step provides an overview of possible approaches to perform the weaving of aspects and base components. One of the weaving approaches is based on modifying the CIL code stored in binary files, called assemblies. The fourth, and last, step is the discussion of the creation of the weaver tool.

The result is the CIL Weaving Tool, which consists of two parts. The first part is the *PE Weaver*, responsible for creating textual IL files from the assemblies and recreating the assemblies after weaving. The second part is the *IL Weaver*, which performs the actual weaving on the textual IL files. To define the weave operations that have to be performed by the IL Weaver, a *weave specification file* has to be supplied. This weave specification file allows the definition of *weave points*, which are based on AOP constructs.

Contents

List of Figures

List of Tables

Nomenclature

AOP	Aspect-Oriented Programming
API	Application Programming Interface
assembly	A compiled and versioned collection of code and metadata that forms a single functional unit, which is shared within the CLR.
boxing	The operation performed on a value type that copies the data from the value into an object of its boxed type allocated on the garbage collected heap.
CIL	Common Intermediate Language
CIL instruction	An instruction or operation that is defined in the CIL, e.g. addition, and subtraction.
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CTS	Common Type System
directive	Basically, a directive is an instruction to the assembler to carry out some unit of work. A directive in the CIL can be recognised by the dot preceeding the directive name.
FCL	Framework Class Library
GUI	Graphical User Interface
IL	Intermediate Language
JIT	Just-in-time
JVM	Java Virtual Machine

manifest	States the name of the assembly, the version, the locale, a list of files that form the assembly, any dependencies the assembly has, and which features are exported.
module	A compiled collection of code without a manifest. Modules can be used to compose assemblies.
MSIL	Microsoft Intermediate Language, specific name for the intermediate language of the .NET Framework, often the general name CIL instead of MSIL is used.
MSIL	Microsoft Intermediate Language
OOP	Object-Oriented Programming
OpCode	Operation Code
opcode	see 'CIL instruction'
PDA	Personal Digital Assistant
PE file	Portable Executable file, .NET assemblies are distributed within PE files.
reference types	Represent class types, array types, pointer types, and interface types. In other words anything that is not a value type.
UML	Unified Modeling Language
unboxing	The operation performed on a value type that returns a pointer to the actual value, i.e. the sequence of bits in memory, held in a boxed object.
value types	Represent the simple or primitive types of many languages, e.g. int or float.
XML	eXtensible Markup Language

CHAPTER 1

Introduction to AOSD

The first two chapters have originally been written by seven M. Sc. students [20, 11, 59, 6, 53, 19, 5] at the University of Twente. The chapters have been rewritten for use in the following theses [58, 9, 56, 22, 10, 21, 52]. They serve as a general introduction into Aspect-Oriented Software Development and Compose* in particular.

1.1 Introduction

The goal of software engineering is to solve a problem by implementing a software system. The things of interest are called concerns. They exist at every level of the engineering process. A recurrent theme in engineering is that of modularization: separation and localization of concerns. The goal of modularization is to create maintainable and reusable software. A programming language is used to implement concerns.

Fifteen years ago the dominant programming language paradigm was procedural programming. This paradigm is characterized by the use of statements that update state variables. Examples are Algol-like languages such as Pascal, C, and Fortran.

Other programming paradigms are the functional, logic, object-oriented, and aspect-oriented paradigms. Figure 1.1 summarizes the dates and ancestry of several important languages [62]. Every paradigm uses a different modularization mechanism for separating concerns into modules.

Functional languages try to solve problems without resorting to variables. These languages are entirely based on functions over lists and trees. Lisp and Miranda are examples of functional languages.

A logic language is based on a subset of mathematical logic. The computer is programmed to infer relationships between values, rather than to compute output values from input values. Prolog is currently the most used logic language [62].

A shortcoming of procedural programming is that global variables can potentially be accessed

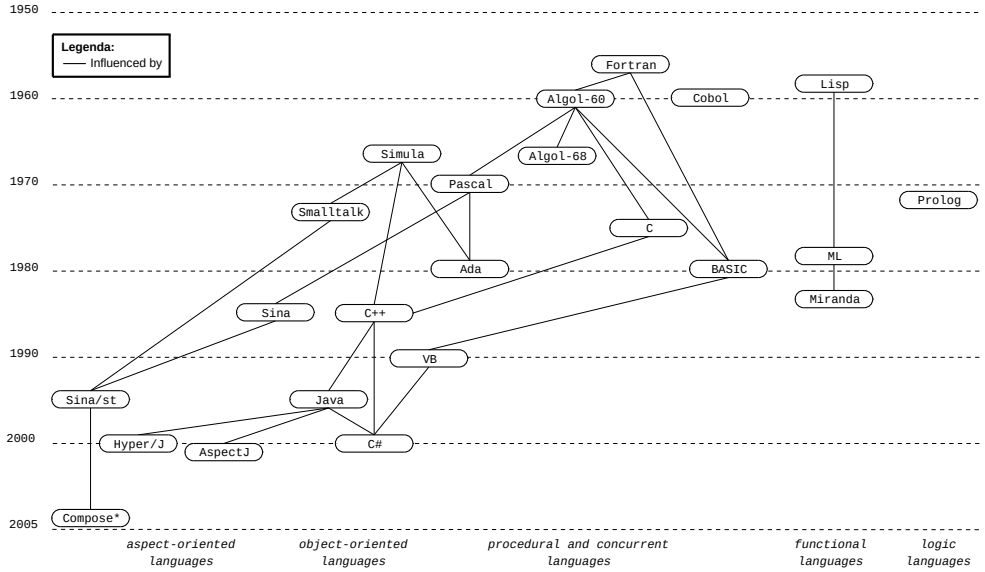


Figure 1.1: Dates and ancestry of several important languages

and updated by any part of the program. This can result in unmanageable programs because no module that accesses a global variable can be understood independently from other modules that also access that global variable.

The Object-Oriented Programming (OOP) paradigm improves modularity by encapsulating data with methods inside objects. The data may only be accessed indirectly, by calling the associated methods. Although the concept appeared in the seventies, it took twenty years to become popular [62]. The most well known object-oriented languages are C++, Java, C#, and Smalltalk.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, adaptability, reusability, and others. They all influence the decomposition, often in conflicting ways [13].

Existing modularization mechanisms typically support only a small set of decompositions and usually only a single dominant modularization at a time. This is known as the tyranny of the dominant decomposition [55]. A specific decomposition limits the ability to implement other concerns in a modular way. For example, OOP modularizes concerns in classes and only fixed relations are possible. Implementing a concern in a class might prevent another concern from being implemented as a class.

Aspect-Oriented Programming (AOP) is a paradigm that solves this problem.

AOP is commonly used in combination with OOP but can be applied to other paradigms as well. The following sections introduce an example to demonstrate the problems that may arise with OOP and show how AOP can solve this. Finally, we look at three particular AOP methodologies in more detail.

```
1  public class Add extends Calculation{
2
3    private int result;
4    private CalcDisplay calcDisplay;
5    private Tracer trace;
6
7    Add() {
8      result = 0;
9      calcDisplay = new CalcDisplay();
10     trace = new Tracer();
11   }
12
13   public void execute(int a, int b) {
14     trace.write("void Add.execute(int, int
         )");
15     result = a + b;
16     calcDisplay.update(result);
17   }
18
19   public int getLastResult() {
20     trace.write("int Add.getLastResult()")
         ;
21     return result;
22   }
23 }
```

(a) Addition

```
1  public class CalcDisplay {
2    private Tracer trace;
3
4    public CalcDisplay() {
5      trace = new Tracer();
6    }
7
8    public void update(int value){
9      trace.write("void CalcDisplay.update(
         int)");
10     System.out.println("Printing new value
          of calculation: "+value);
11   }
12 }
```

(c) CalcDisplay

Listing 1.1: Modeling addition, display, and logging without using aspects

1.2 Traditional Approach

Consider an application containing an object `Add` and an object `CalcDisplay`. `Add` inherits from the abstract class `Calculation` and implements its method `execute(a, b)`. It performs the addition of two integers. `CalcDisplay` receives an update from `Add` if a calculation is finished and prints the result to screen. Suppose all method calls need to be traced. The objects use a `Tracer` object to write messages about the program execution to screen. This is implemented by a method called `write`. Three concerns can be recognized: addition, display, and tracing. The implementation might look something like Listing 1.1.

From our example, we recognize two forms of crosscutting: *code tangling* and *code scattering*.

The addition and display concerns are implemented in classes `Add` and `CalcDisplay` respectively. Tracing is implemented in the class `Tracer`, but also contains code in the other two classes (lines 5, 10, 14, and 20 in (a) and 2, 5, and 9 in (b)). If a concern is implemented across several classes it is said to be scattered. In the example of Listing 1.1 the tracing concern is scattered.

Usually a scattered concern involves code *replication*. That is, the same code is implemented a number of times. In our example the classes `Add` and `CalcDisplay` contain similar tracing code.

In class `Add` the code for the addition and tracing concerns are intermixed. In class `CalcDisplay` the code for the display and tracing concerns are intermixed. If more then one concern is implemented in a single class they are said to be tangled. In our example the addition and tracing concerns are tangled. Also display and tracing concerns are tangled. Crosscutting code

has the following consequences:

Code is difficult to change
: Changing a scattered concern requires us to modify the code in several places. Making modifications to a tangled concern class requires checking for side-effects with all existing crosscutting concerns;

Code is harder to reuse
: To reuse an object in another system, it is necessary to either remove the tracing code or reuse the (same) tracer object in the new system;

Code is harder to understand
: Tangled code makes it difficult to see which code belongs to which concern.

1.3 AOP Approach

To solve the problems with crosscutting, several techniques are being researched that attempt to increase the expressiveness of the OO paradigm. Aspect-Oriented Programming (AOP) introduces a modular structure, the aspect, to capture the location and behavior of crosscutting concerns. Examples of Aspect-Oriented languages are Sina, AspectJ, Hyper/J, and Compose*. A special syntax is used to specify aspects and the way in which they are combined with regular objects. The fundamental goals of AOP are twofold [18]: first to provide a mechanism to express concerns that crosscut other components. Second to use this description to allow for the separation of concerns.

Join points are well-defined places in the structure or execution flow of a program where additional behavior can be attached. The most common join points are method calls. *Pointcuts* describe a set of join points. This allows us to execute behavior at many places in a program by one expression. *Advice* is the behavior executed at a join point.

In the example of Listing 1.2 the class `Add` does not contain any tracing code and only implements the addition concern. Class `CalcDisplay` also does not contain tracing code. In our example the tracing aspect contains all the tracing code. The pointcut `tracedCalls` specifies at which locations tracing code is executed.

The crosscutting concern is explicitly captured in aspects instead of being embedded within the code of other objects. This has several advantages over the previous code.

Aspect code can be changed
: Changing aspect code does not influence other concerns;

Aspect code can be reused
: The coupling of aspects is done by defining pointcuts. In theory, this low coupling allows for reuse. In practice reuse is still difficult;

Aspect code is easier to understand
: A concern can be understood independent of other concerns;

Aspect pluggability
: Enabling or disabling concerns becomes possible.

```
1  public class Add extends Calculation{
2    private int result;
3    private CalcDisplay calcDisplay;
4
5    Add() {
6      result = 0;
7      calcDisplay = new CalcDisplay();
8    }
9
10   public void execute(int a, int b) {
11     result = a + b;
12     calcDisplay.update(result);
13   }
14
15   public int getLastResult() {
16     return result;
17   }
18 }
```

(a) Addition concern

```
1  aspect Tracing {
2    Tracer trace = new Tracer();
3
4    pointcut tracedCalls():
5      call(* (Calculation+).*(..)) ||
6      call(* CalcDisplay.*(..));
7
8    before(): tracedCalls() {
9      trace.write(thisJoinPoint.getSignature()
           .toString());
10   }
11 }
```

(c) Tracing concern

Listing 1.2: Modeling addition, display, and logging with aspects

1.3.1 AOP Composition

AOP composition can be either symmetric or asymmetric. In the symmetric approach every component can be composed with any other component. This approach is followed by e.g. Hyper/J.

In the asymmetric approach, the base program and aspects are distinguished. The base program is composed with the aspects. This approach is followed by e.g. AspectJ (covered in more detail in the next section).

1.3.2 Aspect Weaving

The integration of components and aspects is called *aspect weaving*. There are three approaches to aspect weaving. The first and second approach rely on adding behavior in the program, either by weaving the aspect in the source code, or by weaving directly in the target language. The target language can be intermediate language (IL) or machine code. Examples of IL are Java byte code and Common Intermediate Language (CIL). The remainder of this chapter considers only intermediate language targets. The third approach relies on adapting the virtual machine. Each method is explained briefly in the following sections.

1.3.2.1 Source Code Weaving

The source code weaver combines the original source with aspect code. It interprets the defined aspects and combines them with the original source, generating input for the native compiler. For the native compiler there is no difference between source code with and without aspects. Hereafter the compiler generates an intermediate or machine language output (depending on the compiler-type).

The advantages of using source code weaving are:

High-level source modification
Since all modifications are done at source code level, there is no need to know the target (output) language of the native compiler;

Aspect and original source optimization
First the aspects are woven into the source code and hereafter compiled by the native compiler. The produced target language has all the benefits of the native compiler optimization passes. However, optimizations specific to exploiting aspect knowledge are not possible;

Native compiler portability
The native compiler can be replaced by any other compiler as long as it has the same input language. Replacing the compiler with a newer version or another target language can be done with little or no modification to the aspect weaver.

However, the drawbacks of source code weaving are:

Language dependency
Source code weaving is written explicitly for the syntax of the input language;

Limited expressiveness
Aspects are limited to the expressive power of the source language. For example, when using source code weaving, it is not possible to add multiple inheritance to a single inheritance language.

1.3.2.2 Intermediate Language Weaving

Weaving aspects through an intermediate language gives more control over the executable program and solves some issues as identified in subsubsection 1.3.2.1 on source code weaving. Weaving at this level allows for creating combinations of intermediate language constructs that can not be expressed at the source code level. Although IL can be hard to understand, IL weaving has several advantages over source code weaving:

Programming language independence
All compilers generating the target IL output can be used;

More expressiveness
It is possible to create IL constructs that are not possible in the original programming language;

Source code independence
Can add aspects to programs and libraries without using the source code (which may not be available);

Adding aspects at load- or runtime
A special class loader or runtime environment can decide and do dynamic weaving. The aspect weaver adds a runtime environment into the program. How and when aspects can be added to the program depend on the implementation of the runtime environment.

However, IL weaving also has drawbacks that do not exist for source code weaving:

Hard to understand
Specific knowledge about the IL is needed;

More error-prone
Compiler optimization may cause unexpected results. Compiler can remove code that breaks the attached aspect (e.g., inlining of methods).

1.3.2.3 Adapting the Virtual Machine

Adapting the virtual machine (VM) removes the need to weave aspects. This technique has the same advantages as intermediate language weaving and can also overcome some of its disadvantages as mentioned in subsubsection 1.3.2.2. Aspects can be added without recompilation, redeployment, and restart of the application [45, 46].

Modifying the virtual machine also has its disadvantages:

Dependency on adapted virtual machines
Using an adapted virtual machine requires that every system should be upgraded to that version;
Virtual machine optimization
People have spend a lot of time optimizing virtual machines. By modifying the virtual machine these optimizations should be revisited. Reintegrating changes introduced by newer versions of the original virtual machine, might have substantial impact.

1.4 AOP Solutions

As the concept of AOP has been embraced as a useful extension to classic programming, different AOP solutions have been developed. Each solution has one or more implementations to demonstrate how the solution is to be used. As described by [12] these differ primarily in:

How aspects are specified
Each technique uses its own aspect language to describe the concerns;
Composition mechanism
Each technique provides its own composition mechanisms;
Implementation mechanism
Whether components are determined statically at compile time or dynamically at run time, the support for verification of compositions, and the type of weaving.
Use of decoupling
Should the writer of the main code be aware that aspects are applied to his code;
Supported software processes
The overall process, techniques for reusability, analyzing aspect performance of aspects, is it possible to monitor performance, and is it possible to debug the aspects.

This section will give a short introduction to AspectJ [26] and Hyperspaces [43], which together with Composition Filters [4] are three main AOP approaches.

1.4.1 AspectJ Approach

AspectJ [26] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP at the moment, and it is finding its way into the industrial

```
1  aspect DynamicCrosscuttingExample {
2    Log log = new Log();
3
4    pointcut traceMethods():
5      execution(edu.utwente.trese.*.*(..));
6
7    before() : traceMethods {
8      log.write("Entering " + thisJointPoint.getSignature());
9    }
10
11   after() : traceMethods {
12     log.write("Exiting " + thisJointPoint.getSignature());
13   }
14 }
```

Listing 1.3: Example of dynamic crosscutting in AspectJ

software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an open Eclipse project. The popularity of AspectJ comes partly from the various extensions based on it, build by several research groups. There are various projects that are porting AspectJ to other languages, resulting in tools such as AspectR and AspectC.

One of the main goals in the design of AspectJ is to make it a compatible extension to Java. AspectJ tries to be compatible in four ways:

Upward compatibility
: All legal Java programs must be legal AspectJ programs;

Platform compatibility
: All legal AspectJ programs must run on standard Java virtual machines;

Tool compatibility
: It must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools and design tools;

Programmer compatibility
: Programming with AspectJ must feel like a natural extension of programming with Java.

AspectJ extends Java with support for two kinds of crosscutting functionality. The first allows defining additional behavior to run at certain well-defined points in the execution of the program and is called the *dynamic crosscutting mechanism*. The other is called the *static crosscutting mechanism* and allows modifying the static structure of classes (methods and relationships between classes). The units of crosscutting implementation are called aspects. An example of an aspect specified in AspectJ is shown in Listing 1.3.

The points in the execution of a program where the crosscutting behavior is inserted are called *join points*. A *pointcut* has a set of join points. In Listing 1.3 is `traceMethods` an example of a pointcut definition. The pointcut includes all executions of any method that is in a class contained by package `edu.utwente.trese`.

The code that should execute at a given join point is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*, *after* and *around* advice that specifies where the additional code is to be inserted. In the example both before

```
1  aspect StaticCrosscuttingExample {
2    private int Log.trace(String traceMsg) {
3      Log.write(" --- MARK --- " + traceMsg);
4    }
5  }
```

Listing 1.4: Example of static crosscutting in AspectJ

and after advice are declared to run at the join points specified by the `traceMethods` pointcut.

Aspects can contain anything permitted in class declarations including definitions of pointcuts, advice and static crosscutting. For example, static crosscutting allows a programmer to add fields and methods to certain classes as shown in Listing 1.4.

The shown construct is called inter-type member declaration and adds a method `trace` to class `Log`. Other forms of inter-type declarations allow developers to declare the parents of classes (superclasses and realized interfaces), declare where exceptions need to be thrown, and allow a developer to define the precedence among aspects.

With its variety of possibilities AspectJ can be considered a useful approach for realizing software requirements.

1.4.2 Hyperspaces Approach

The *Hyperspaces* approach is developed by H. Ossher and P. Tarr at the IBM T.J. Watson Research Center. The Hyperspaces approach adopts the principle of multi-dimensional separation of concerns [43], which involves:

- Multiple, arbitrary dimensions of concerns;
- Simultaneous separation along these dimensions;
- Ability to dynamically handle new concerns and new dimensions of concern as they arise throughout the software life cycle;
- Overlapping and interacting concerns. It is appealing to think of many concerns as independent or orthogonal, but they rarely are in practice.

We explain the Hyperspaces approach by an example written in the *Hyper/J* language. Hyper/J is an implementation of the Hyperspaces approach for Java. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. Hyper/J uses bytecode weaving on binary Java class files and generates new class files to be used for execution. Although the Hyper/J project seems abandoned and there has not been any update in the code or documentation for a while, we still mention it because the Hyperspaces approach offers a unique AOP solution.

As a first step, developers create hyperspaces by specifying a set of Java class files that contain the code units that populate the hyperspace. To do this is, you create a hyperspace specification, as demonstrated in Listing 1.5.

Hyper/J will automatically create a hyperspace with one dimension—the class file dimension. A dimension of concern is a set of concerns that are disjoint. The initial hyperspace will contain all units within the specified package. To create a new dimension you can specify concern

```
1 Hyperspace Pacman
2   class edu.utwente.trese.pacman.*;
```

Listing 1.5: Creation of a hyperspace

mappings, which describe how existing units in the hyperspace relate to concerns in that dimension, as demonstrated in Listing 1.6.

The first line indicates that, by default, all of the units contained within the package `edu.utwente.trese.pacman` address the kernel concern of the feature dimension. The other mappings specify that any method named `trace` or `debug` address the logging and debugging concern respectively. These later mappings override the first one.

Hypermodules are based on concerns and consist of two parts. The first part specifies a set of hyperslices in terms of the concerns identified in the concern matrix. The second part specifies the integration relationships between the hyperslices. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

Listing 1.7 shows a hypermodule with two concerns, kernel and logging. They are related by a `mergeByName` integration relationship. This means that units in the different concerns correspond if they have the same name (`ByName`) and that these corresponding units are to be combined (`merge`). For example, all members of the corresponding classes are brought together into the composed class. The hypermodule results in a hyperslice that contains all the classes without the debugging feature; thus no `debug` methods will be present.

The most important feature of the hyperspaces approach is the support for on-demand remodularisation: the ability to extract hyperslices to encapsulate concerns that were not separated in the original code. Which makes hyperspaces especially useful for evolution of existing software.

1.4.3 Composition Filters

Composition Filters is developed by M. Akşit and L. Bergmans at the TRESE group, which is a part of the Department of Computer Science of the University of Twente, The Netherlands. The composition filters (CF) model predates aspect-oriented programming. It started out as an extension to the object-oriented model and evolved into an aspect-oriented model. The current implementation of CF is Compose⋆, which covers .NET, Java, and C.

One of the key elements of CF is the *message*, a message is the interaction between objects, for instance a method call. In object-oriented programming the message is considered an abstract concept. In the implementations of CF it is therefore necessary to reify the message. This *reified message* contains properties, like where it is send to and where it came from.

```
1 package edu.utwente.trese.pacman: Feature.Kernel
2 operation trace: Feature.Logging
3 operation debug: Feature.Debugging
```

Listing 1.6: Specification of concern mappings

```
1 hypermodule Pacman_Without_Debugging
2   hyperslices: Feature.Kernel, Feature.Logging;
3   relationships: mergeByName;
4 end hypermodule;
```

Listing 1.7: Defining a hypermodule

The concept of CF is that messages that enter and exit an object can be intercepted and manipulated, modifying the original flow of the message. To do so, a layer called the *interface part* is introduced in the CF model, this layer can have several properties. The interface part can be placed on an object, which behavior needs to be altered, and this object is referred to as *inner*.

There are three key elements in CF: messages, filters, and superimposition. Messages are sent from one object to another, if there is an interface part placed on the receiver, then the message that is sent goes through the input filters. In the filters the message can be manipulated before it reaches the inner part, the message can even be sent to another object. How the message will be handled depends on the filter type. An output filter is similar to an input filter, the only difference is that it manipulates messages that originate from the inner part. The latest addition to CF is superimposition, which is used to specify which interfaces needs to be superimposed on which inner objects.

CHAPTER 2

Compose⋆

Compose⋆ is an implementation of the composition filters approach. There are three target environments: the .NET, Java, and C. This chapter is organized as follows, first the evolution of Composition Filters and its implementations are described, followed by an explanation of the Compose⋆ language and a demonstrating example. In the third section, the Compose⋆ architecture is explained, followed by a description of the features specific to Compose⋆.

2.1 Evolution of Composition Filters

Compose⋆ is the result of many years of research and experimentation. The following time line gives an overview of what has been done in the years before and during the Compose⋆ project.

1985 The first version of Sina is developed by Mehmet Akşit. This version of Sina contains a preliminary version of the composition filters concept called semantic networks. The semantic network construction serves as an extension to objects, such as classes, messages, or instances. These objects can be configured to form other objects such as classes from which instances can be created. The object manager takes care of synchronization and message processing of an object. The semantic network construction can express key concepts like delegation, reflection, and synchronization [27].

1987 Together with Anand Tripathi of the University of Minnesota the Sina language is further developed. The semantic network approach is replaced by declarative specifications and the interface predicate construct is added.

1991 The interface predicates are replaced by the dispatch filter, and the wait filter manages the synchronization functions of the object manager. Message reflection and real-time specifications are handled by the meta filter and the real-time filter [3].

1995 The Sina language with Composition Filters is implemented using Smalltalk [27]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [15].

```
1   filtermodule{
2     internals
3     externals
4     conditions
5     inputfilters
6     outputfilters
7   }
8
9   superimposition{
10    selectors
11    filtermodules
12    annotations
13    constraints
14  }
15
16  implementation
17 }
```

Listing 2.1: Abstract concern template

1999 The composition filters language ComposeJ [63] is developed and implemented. The implementation consists of a preprocessor capable of translating composition filter specifications into the Java language.

2001 ConcernJ is implemented as part of a M. Sc. thesis [49]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules and to facilitate crosscutting concerns.

2003 The start of the Compose⋆ project, the project is described in further detail in this chapter.

2004 The first release of Compose⋆, based on .NET.

2005 The start of the Java port of Compose⋆.

2006 Porting Compose⋆ to C is started.

2.2 Composition Filters in Compose⋆

A Compose⋆ application consists of concerns that can be divided in three parts: filter module specification, superimposition, and implementation. A filter module contains the filter logic to filter on messages that are incoming or outgoing the superimposed object. A message has a target, which is an object reference, and a selector, which is a method name. The superimposition part specifies which filter modules, annotations, conditions, and methods need to be superimposed on which objects. The implementation part contains the class implementation of the concern. How these parts are placed in a concern is shown in Listing 2.1.

The working of the filter module is shown in Figure 2.1. A filter module can contain input and output filters. The difference between these two sets of filters is that the first is used to filter on incoming messages and the second filter set is used on the outgoing messages. A return of a method is not considered as an outgoing message. A filter has three parts: the filter identifier, the filter type, and one or more filter elements. The filter element exist out of an optional condition part, a matching part, and a substitution part. These parts are shown below:

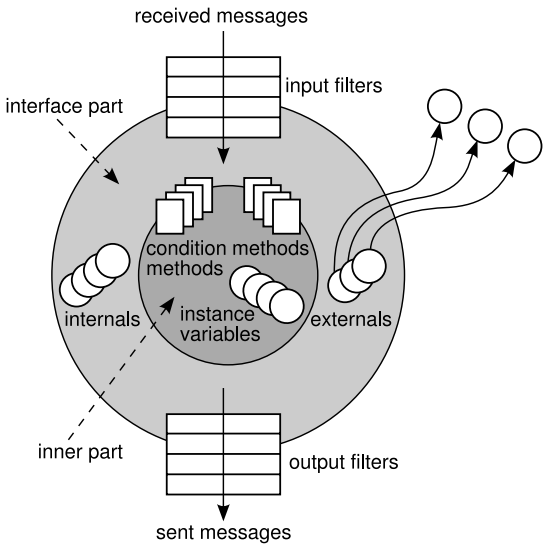


Figure 2.1: Components of the composition filters model

$$\overbrace{stalker\_filter}^{identifier} : \overbrace{Dispatch}^{filter\ type} = \{\overbrace{!pacmanIsEvil}^{condition\ part} => \overbrace{[*.getNextMove]}^{matching\ part} \overbrace{stalk\_strategy.getNextMove}^{substitution\ part} \}$$

The filter identifier is the unique name for a filter in a filter module. A filter matches when both the condition as the matching provide the boolean value true. In the demonstrated filter it matches on every message where the selector is `getNextMove`, the '*' in the target means that every target matches. When the condition part and the matching part are true, the message is substituted with the values of the substitution part. How these values are substituted and how the message continues depends on the filter type. At the moment there are four basic filter types in Compose⋆; it is possible to write custom filter types.

Dispatch
: If the message is accepted, it is dispatched to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for input filters;

Send
: If the message is accepted, it is sent to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for output filters;

Error
: If the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set;

Meta
: If the message is accepted, the message is sent as a parameter of another meta message to an internal or external object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message and can re-activate the execution of the message.

The `pacmanIsEvil` used in the condition part must be declared in the conditions section of a filtermodule. The targets that are used in a filter must declared as internals or externals. Internals are objects which are unique for each instance of a filter module and externals are shared between filter modules.

The filter modules can be superimposed on classes with filter module binding, this binding has a selection of objects on one side and a filter module on the other side. The selection is defined with a selector definition. The selector uses predicates, such as `isClassWithNameInList`, `isNamespaceWithName`, and `namespaceHasClass`, to select objects. It is also possible to bind conditions, methods, and annotations to classes with the use of superimposition.

The last part of the concern is the implementation part. In the implementation part we can define the object behavior of the concern, so for example in a logging concern, we can define specific log functions.

2.3 Demonstrating Example

To illustrate the Compose⋆ toolset, this section introduces a *Pacman* example. The Pacman game is a classic arcade game in which the user, represented by pacman, moves in a maze to eat vitamins. Meanwhile, a number of ghosts try to catch and eat pacman. There are, however, four mega vitamins in the maze that make pacman evil. In its evil state, pacman can eat ghosts. A simple list of requirements for the Pacman game is briefly discussed here:

- The number of lives taken from pacman when eaten by a ghost;
- A game should end when pacman has no more lives;
- The score of a game should increase when pacman eats a vitamin or a ghost;
- A user should be able to use a keyboard to move pacman around the maze;
- Ghosts should know whether pacman is evil or not;
- Ghosts should know where pacman is located;
- Ghosts should, depending on the state of pacman, hunt or flee from pacman.

2.3.1 Initial Object-Oriented Design

Figure 2.2 shows an initial object-oriented design for the Pacman game. Note that this UML class diagram does not show the trivial accessors. The classes in this diagram are:

Game
: This class encapsulates the control flow and controls the state of a game;

Ghost
: This class is a representation of a ghost chasing pacman. Its main attribute is a property that indicates whether it is scared or not (depending on the evil state of pacman);

GhostView
: This class is responsible for painting ghosts;

Glyph
: This is the superclass of all mobile objects (pacman and ghosts). It contains common information like direction and speed;

Keyboard
: This class accepts all keyboard input and makes it available to pacman;

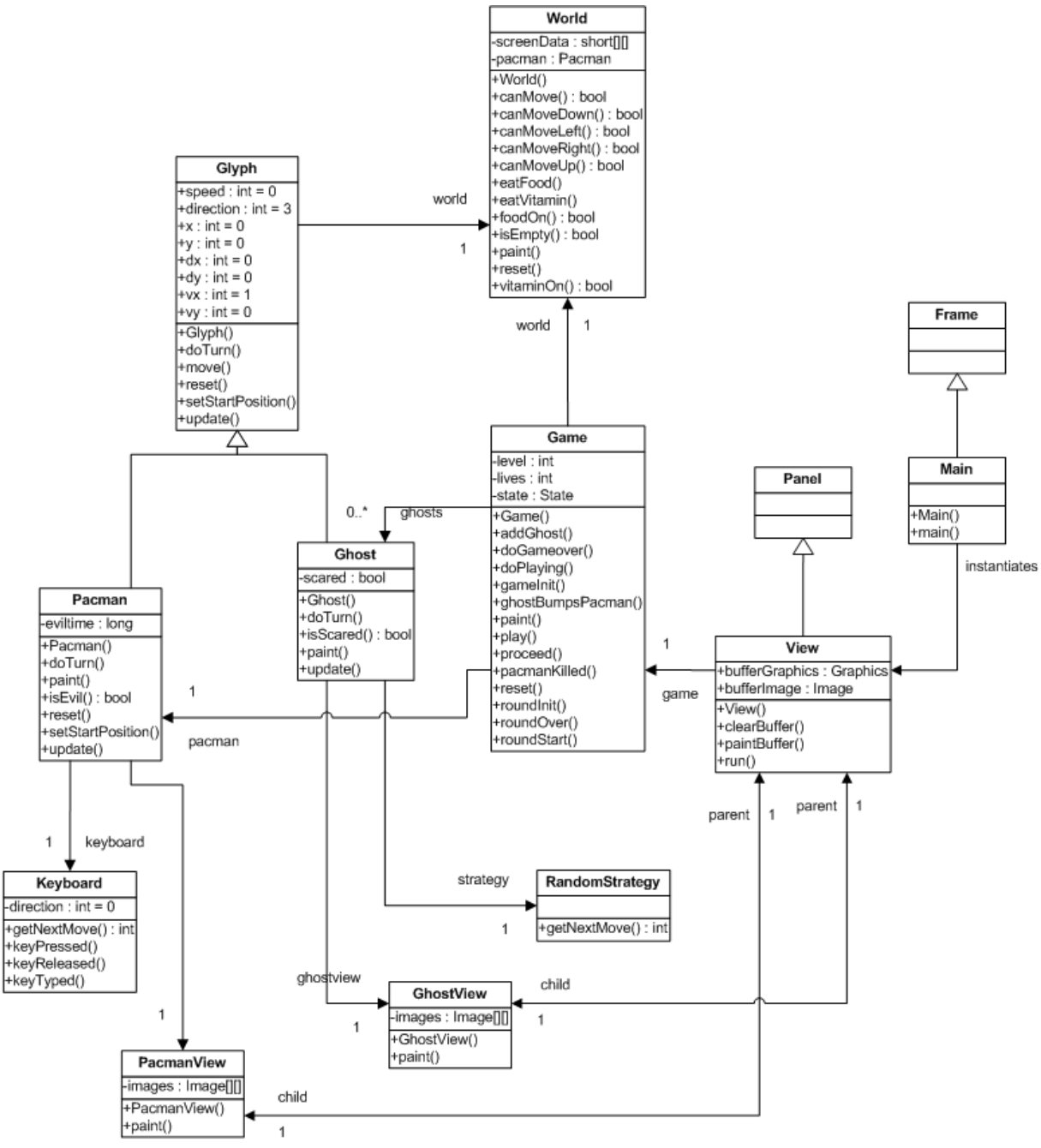


Figure 2.2: UML class diagram of the object-oriented Pacman game

Main
This is the entry point of a game;

Pacman
This is a representation of the user controlled element in the game. Its main attribute is a property that indicates whether pacman is evil or not;

PacmanView
This class is responsible for painting pacman;

RandomStrategy
By using this strategy, ghosts move in random directions;

View
This class is responsible for painting a maze;

World
This class has all the information about a maze. It knows where the vitamins, mega vitamins and most importantly the walls are. Every class derived from class `Glyph` checks whether movement in the desired direction is possible.

2.3.2 Completing the Pacman Example

The initial object-oriented design, described in the previous section, does not implement all the stated system requirements. The missing requirements are:

- The application does not maintain a score for the user;
- Ghosts move in random directions instead of chasing or fleeing from pacman.

In the next sections, we describe why and how to implement these requirements in the Compose⋆ language.

2.3.2.1 Implementation of Scoring

The first system requirement that we need to add to the existing Pacman game is scoring. This concern involves a number of events. First, the score should be set to zero when a game starts. Second, the score should be updated whenever pacman eats a vitamin, mega vitamin or ghost. And finally, the score itself has to be painted on the maze canvas to relay it back to the user. These events scatter over multiple classes: `Game` (initializing score), `World` (updating score), `Main` (painting score). Thus scoring is an example of a crosscutting concern.

To implement scoring in the Compose⋆ language, we divide the implementation into two parts. The first part is a Compose⋆ concern definition stating which filter modules to superimpose. Listing 2.2 shows an example Compose⋆ concern definition of scoring.

This concern definition is called `DynamicScoring` (line 1) and contains two parts. The first part is the declaration of a filter module called `dynamicscoring` (lines 2–11). This filter module contains one *meta filter* called `score_filter` (line 6). This filter intercepts five relevant calls and sends the message in a reified form to an instance of class `Score`. The final part of the concern definition is the superimposition part (lines 12–18). This part defines that the filter module `dynamicscoring` is to be superimposed on the classes `World`, `Game` and `Main`.

The final part of the scoring concern is the so-called *implementation part*. This part is defined by a class `Score`. Listing 2.3 shows an example implementation of class `Score`. Instances of this

```
1  concern DynamicScoring in pacman {
2    filtermodule dynamicscoring {
3      externals
4        score : pacman.Score = pacman.Score.instance();
5      inputfilters
6        score_filter : Meta = {[*.eatFood] score.eatFood,
7                               [*.eatGhost] score.eatGhost,
8                               [*.eatVitamin] score.eatVitamin,
9                               [*.gameInit] score.initScore,
10                              [*.setForeground] score.setupLabel}
11   }
12   superimposition {
13     selectors
14       scoring = { C | isClassWithNameInList(C, ['pacman.World',
15                                  'pacman.Game', 'pacman.Main']) };
16     filtermodules
17       scoring <- dynamicscoring;
18   }
19 }
```

Listing 2.2: `DynamicScoring` concern in Compose⋆

class receive the messages sent by `score_filter` and subsequently perform the events related to the scoring concern. In this way, all scoring events are encapsulated in one class and one Compose⋆ concern definition.

2.3.2.2 Implementation of Dynamic Strategy

The last system requirement that we need to implement is the dynamic strategy of ghosts. This means that a ghost should, depending on the state of pacman, hunt or flee from pacman. We can implement this concern by using the strategy design pattern. However, in this way, we need to modify the existing code. This is not the case when we use Compose⋆ *dispatch filters*. Listing 2.4 demonstrates this.

This concern uses *dispatch* filters to intercept calls to method `RandomStrategy.getNextMove` and redirect them to either `StalkerStrategy.getNextMove` or `FleeStrategy.getNextMove`. If pacman is not evil, the intercepted call matches the first filter, which dispatches the intercepted call to method `StalkerStrategy.getNextMove` (line 9). Otherwise, the intercepted call matches the second filter, which dispatches the intercepted call to method `FleeStrategy.getNextMove` (line 11).

2.4 Compose⋆ Architecture

An overview of the Compose⋆ architecture is illustrated in Figure 2.3. The Compose⋆ architecture can be divided in four layers [42]: IDE, compile time, adaptation, and runtime.

2.4.1 Integrated Development Environment

Some of the purposes of the Integrated Development Environment (IDE) layer are to interface with the native IDE and to create a build configuration. In the build configuration it is specified

```
1  import Composestar.Runtime.FLIRT.message.*;
2  import java.awt.*;
3
4  public class Score
5  {
6    private int score = -100;
7    private static Score theScore = null;
8    private Label label = new java.awt.Label("Score: 0");
9
10   private Score() {}
11
12   public static Score instance() {
13     if(theScore == null) {
14       theScore = new Score();
15     }
16     return theScore;
17   }
18
19   public void initScore(ReifiedMessage rm) {
20     this.score = 0;
21     label.setText("Score: "+score);
22   }
23
24   public void eatGhost(ReifiedMessage rm) {
25     score += 25;
26     label.setText("Score: "+score);
27   }
28
29   public void eatVitamin(ReifiedMessage rm) {
30     score += 15;
31     label.setText("Score: "+score);
32   }
33
34   public void eatFood(ReifiedMessage rm) {
35     score += 5;
36     label.setText("Score: "+score);
37   }
38
39   public void setupLabel(ReifiedMessage rm) {
40     rm.proceed();
41     label = new Label("Score: 0");
42     label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
43     Main main = (Main)Composestar.Runtime.FLIRT.message.MessageInfo
44                                  .getMessageInfo().getTarget();
45     main.add(label,BorderLayout.SOUTH);
46   }
47 }
```

Listing 2.3: Implementation of class `Score`

```
concern DynamicStrategy in pacman {
  filtermodule dynamicstrategy {
    internals
      stalk_strategy : pacman.Strategies.StalkerStrategy;
      flee_strategy : pacman.Strategies.FleeStrategy;
    conditions
      pacmanIsEvil : pacman.Pacman.isEvil();
    inputfilters
      stalker_filter : Dispatch = {!pacmanIsEvil =>
                        [*.getNextMove] stalk_strategy.getNextMove};
      flee_filter : Dispatch = {
                        [*.getNextMove] flee_strategy.getNextMove}
  }
  superimposition {
    selectors
      random = { C | isClassWithName(C,
                        'pacman.Strategies.RandomStrategy') };
    filtermodules
      random <- dynamicstrategy;
  }
}
```

Listing 2.4: `DynamicStrategy` concern in Compose⋆

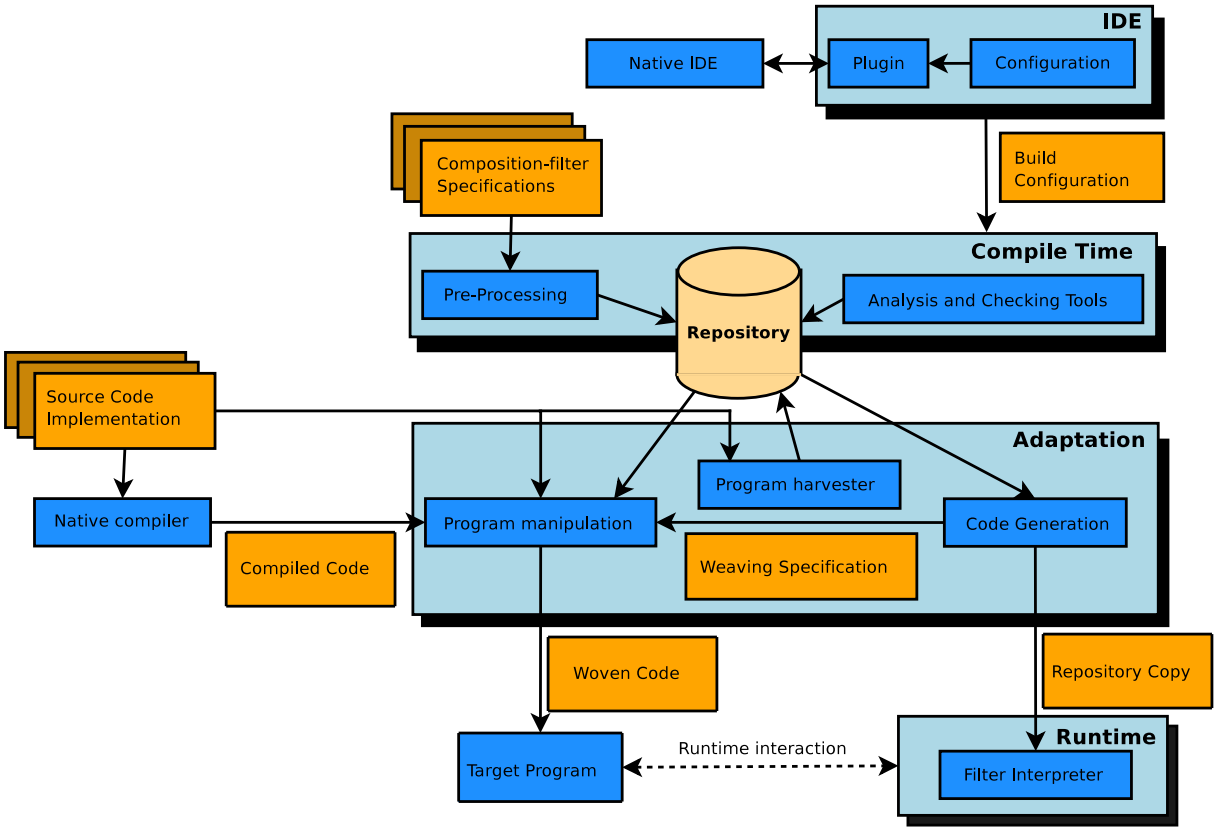


Figure 2.3: Overview of the Compose⋆ architecture

which source files and settings are required to build a Compose⋆ application. After creating the build configuration the compile time is started.

The creation of a build configuration can be done manually or by using a plug-in. Examples of these plug-ins are the Visual Studio add-in for Compose⋆/.NET and the Eclipse plug-in for Compose⋆/J and Compose⋆/C.

2.4.2 Compile Time

The compile time layer is platform independent and reasons about the correctness of the composition filter implementation with respect to the program which allows the target program to be build by the adaptation.

The compile time 'pre-processes' the composition filter specifications by parsing the specification, resolving the references, and checking its consistency. To provide an extensible architecture to facilitate this process a blackboard architecture is chosen. This means that the compile time uses a general knowledgebase that is called the 'repository'. This knowledgebase contains the structure and metadata of the program which different modules can execute their activities on. Examples of modules within analysis and validation are the three modules SANE, LOLA and FILTH. These three modules are responsible for (some) of the analysis and validation of the super imposition and its selectors.

2.4.3 Adaptation

The adaptation layer consists of the program manipulation, harvester, and code generator. These components connect the platform independent compile time to the target platform. The harvester is responsible for gathering the structure and the annotations within the source program and adding this information to the knowledgebase. The code generation generates a reduced copy of the knowledgebase and the weaving specification. This weaving specification is then used by the weaver contained by the program manipulation to weave in the calls to the runtime into the target program. The end result of the adaptation the target program which interfaces wit the runtime.

2.4.4 Runtime

The runtime layer is responsible for executing the concern code at the join points. It is activated at the join points by function calls that are woven in by the weaver. A reduced copy of the knowledgebase containing the necessary information for filter evaluation and execution is enclosed with the runtime. When the function is filtered the filter is evaluated. Depending on if the the condition part evaluates to true, and the matching part matches the accept or reject behavior of the filter is executed. The runtime also facilitates the debugging of the composition filter implementations.

2.5 Platforms

Compose⋆ can in theory be applied to any programming language given certain assumptions are met. Currently Compose⋆ has three platforms.

2.5.1 Java

Compose⋆/J, the Java platform of Compose⋆, uses different compiling and weaving tools then the other platforms. For the use of Compose⋆/J an Eclipse plug-in is provided.

2.5.2 C

Compose⋆/C, the C platform of Compose⋆, is different from its Java and .NET counterparts because it does not have a runtime interpreter. This implies that the filters implementation of Compose⋆/C uses generated composition filter code that is weaved directly in the source code. Because the programming language C does not have the concept of objects the reasoning within Compose⋆ is based on sets of functions. Like the Java platform, Compose⋆/C provides a plug-in for Eclipse.

2.5.3 .NET

The .NET platform called Compose⋆/.NET of Compose⋆ is the oldest implementation of Compose⋆. Because Compose⋆/.NET works with CIL code, it is programming language independent as long as the programming language can be compiled to CIL code. The .NET platform uses a Visual Studio add-in for ease of development.

2.6 Features Specific to Compose⋆

The Composition Filters approach uses a restricted (pattern matching) language to define filters. This language makes it possible to reason about the semantics of the concern. Compose⋆ offers three features that use this possibility, which originate in more control and correctness over an application under construction. These features are:

Ordering of filter modules
: It is possible to specify how the superimposition of filter modules should be ordered. Ordering constraints can be specified in a fixed, conditional, or partial manner. A fixed ordering can be calculated exactly, whereas a conditional ordering is dependent on the result of filter execution and therefore evaluated at runtime. When there are multiple valid orderings of filtermodules on a join point, partial ordering constraints can be applied to reduce this number. These constraints can be declared in the concern definition;

Filter consistency checking
: When superimposition is applied, Compose⋆ is able to detect if the ordering and conjunction of filters creates a conflict. For example, imagine a set of filters where the first filter only evaluates method *m* and another filter only evaluates methods *a* and *b*. In this case the latter filter is only reached with method *m*; this is consequently rejected and as a result the superimposition may never be executed. There are different scenarios that lead to these kinds of problems, e.g., conditions that exclude each other;

Reason about semantic problems
: When multiple pieces of advice are added to the same join point, Compose⋆ can reason about problems that may occur. An example of such a conflict is the situation where a

real-time filter is followed by a wait filter. Because the wait filter can wait indefinitely, the real-time property imposed by the real-time filter may be violated.

The above mentioned conflict analyzers all work on the assumption that the behavior of every filter is well-defined. This is not the case for the meta filter, its user-undefined, and therefore unpredictable, behavior poses a problem to the analysis tools.

Furthermore, Compose⋆ is extended with features that enhance the usability. These features are briefly described below:

Integrated Development Environment support
The Compose⋆ implementations all have a IDE plug-in; Compose⋆/.NET for Visual Studio, Compose⋆/J and Compose⋆/C for Eclipse;

Debugging support
The debugger shows the flow of messages through the filters. It is possible to place breakpoints to view the state of the filters;

Incremental building process
When a project is build and not all the modules are changed, incremental building saves time.

Some language properties of Compose⋆ can also be seen as features, being:

Language independent concerns
A Compose⋆ concern can be used for all the Compose⋆ platforms, because the composition filters approach is language independent;

Reusable concerns
The concerns are easy to reuse, through the dynamic filter modules and the selector language;

Expressive selector language
Program elements of an implementation language can be used to select a set of objects to superimpose on;

Support for annotations
Using the selector, annotations can be woven at program elements. At the moment annotations can be used for superimposition.

CHAPTER 3

Introduction to the .NET Framework

This chapter gives an introduction to the .NET Framework of Microsoft. First, the architecture of the .NET Framework is introduced. This section includes terms like the Common Language Runtime, the .NET Class Library, the Common Language Infrastructure and the Intermediate Language. These are discussed in more detail in the sections following the architecture.

3.1 Introduction

Microsoft defines [35] .NET as follows; ".NET is the Microsoft Web services strategy to connect information, people, systems, and devices through software.". There are different .NET technologies in various Microsoft products providing the capabilities to create solutions using web services. Web services are small, reusable applications that help computers from many different operating system platforms work together by exchanging messages. Based on industry standards like XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), and WSDL (Web Services Description Language) they provide a platform and language independent way to communicate.

Microsoft products, such as Windows Server System (providing web services) or Office System (using web services) are some of the .NET technologies. The technology described in this chapter is the .NET Framework. Together with Visual Studio, an integrated development environment, they provide the developer tools to create programs for .NET.

Many companies are largely dependent on the .NET Framework, but need or want to use AOP. Currently there is no direct support for this in the Framework. The Compose⋆/.NET project is addressing these needs with its implementation of the Composition Filters approach for the .NET Framework.

This specific Compose⋆ version for .NET has two main goals. First, it combines the .NET Framework with AOP through Composition Filters. Second, Compose⋆ offers superimposition in a language independent manner. The .NET Framework supports multiple languages and is, as such, suitable for this purpose. Composition Filters are an extension of the object-oriented

mechanism as offered by .NET, hence the implementation is not restricted to any specific object-oriented language.

3.2 Architecture of the .NET Framework

The .NET Framework is Microsoft's platform for building, deploying, and running Web Services and applications. It is designed from scratch and has a consistent API providing support for component-based programs and Internet programming. This new Application Programming Interface (API) has become an integral component of Windows. The .NET Framework was designed to fulfill the following objectives [32]:

Consistency
: Allow object code to be stored and executed locally, executed locally but Internet-distributed, or executed remotely and to make the developer experience consistent across a wide variety of types of applications, such as Windows-based applications and Web-based applications;

Operability
: The ease of operation is enhanced by minimizing versioning conflicts and providing better software deployment support;

Security
: All the code is executed safely, including code created by an unknown or semi-trusted third party;

Efficiency
: The .NET Framework compiles applications to machine code before running thus eliminating the performance problems of scripted or interpreted environments;

Interoperability
: Code based on the .NET Framework can integrate with other code because all communication is built on industry standards.

The .NET Framework consists of two main components [32]: the Common Language Runtime (CLR, simply called the .NET Runtime or Runtime for short) and the .NET Framework Class Library (FCL). The CLR is the foundation of the .NET Framework, executing the code and providing the core services such as memory management, thread management and exception handling. The CLR is described in more detail in section 3.3. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications such as Web Forms and XML Web services. section 3.5 describes the class libraries in more detail.

The code run by the runtime is in a format called Common Intermediate Language (CIL), further explained in section 3.6. The Common Language Infrastructure (CLI) is an open specification that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework. section 3.4 tells more about this specification.

Figure 3.1 shows the relationship of the .NET Framework to other applications and to the complete system. The two parts, the class library and the runtime, are managed, i.e., applications managed during execution. The operating system is in the core, managed and unmanaged applications operate on the hardware. The runtime can us other object libraries and the class library, but the other libraries can use the same class library them self.

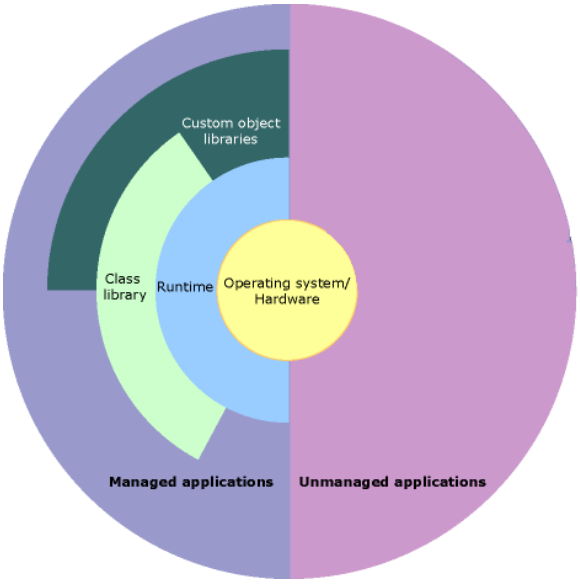


Figure 3.1: Context of the .NET Framework (Modified) [32]

Besides the Framework, Microsoft also provides a developer tool called the Visual Studio. This is an IDE with functionality across a wide range of areas allowing developers to build applications with decreased development time in comparison with developing applications using command line compilers.

3.2.1 Version 2.0 of .NET

In November 2005, Microsoft released a successor of the .NET Framework. Major changes are the support for generics, the addition of nullable types, 64 bit support, improvements in the garbage collector, new security features and more network functionality.

Generics make it possible to declare and define classes, structures, interfaces, methods and delegates with unspecified or generic type parameters instead of specific types. When the generic is used, the actual type is specified. This allows for type-safety at compile-time. Without generics, the use of casting or boxing and unboxing decreases performance. By using a generic type, the risks and costs of these operations is reduced.

Nullable types allow a value type to have a normal value or a null value. This null value can be useful for indicating that a variable has no defined value because the information is not currently available.

Besides changes in the Framework, there are also improvements in the four main Microsoft .NET programming languages (C#, VB.NET, J# and C++). The language elements are now almost equal for all languages. For instance, additions to the Visual Basic language are the support for unsigned values and new operators and additions to the C# language include the ability to define anonymous methods thus eliminating the need to create a separate method.

A new Visual Studio 2005 edition was released to support the new Framework and functionalities to create various types of applications.

3.3 Common Language Runtime

The Common Language Runtime executes code and provides core services. These core services are memory management, thread execution, code safety verification and compilation. Apart from providing services, the CLR also enforces code access security and code robustness. Code access security is enforced by providing varying degrees of trust to components, based on a number of factors, e.g., the origin of a component. This way, a managed component might or might not be able to perform sensitive functions, like file-access or registry-access. By implementing a strict type-and-code-verification infrastructure, called the Common Type System (CTS), the CLR enforces code robustness. Basically there are two types of code;

Managed
: Managed code is code, which has its memory handled and its types validated at execution by the CLR. It has to conform to the Common Type Specification (CTS section 3.4). If interoperability with components written in other languages is required, managed code has to conform to an even more strict set of specifications, the Common Language Specification (CLS). The code is run by the CLR and is typically stored in an intermediate language format. This platform independent intermediate language is officially known as Common Intermediate Language (CIL section 3.6) [60].

Unmanaged
: Unmanaged code is not managed by the CLR. It is stored in the native machine language and is not run by the runtime but directly by the processor.

All language compilers (targeting the CLR) generate managed code (CIL) that conforms to the CTS.

At runtime, the CLR is responsible for generating platform specific code, which can actually be executed on the target platform. Compiling from CIL to the native machine language of the platform is executed by the just-in-time (JIT) compiler. Because of this language independent layer it allows the development of CLRs for any platform, creating a true interoperability infrastructure [60]. The .NET Runtime from Microsoft is actually a specific CLR implementation for the Windows platform. Microsoft has released the *.NET Compact Framework* especially for devices such as personal digital assistants (PDAs) and mobile phones. The .NET Compact Framework contains a subset of the normal .NET Framework and allows .NET developer to write mobile applications. Components can be exchanged and web services can be used so an easier interoperability between mobile devices and workstations/servers can be implemented [34].

At the time of writing, the .NET Framework is the only advanced Common Language Infrastructure (CLI) implementation available. A shared-source[1] implementation of the CLI for research and teaching purposes was made available by Microsoft in 2002 under the name Rotor [54]. In 2006 Microsoft released an updated version of Rotor for the .NET platform version two. Also Ximian is working on an open source implementation of the CLI under the name Mono[2], targeting both Unix/Linux and Windows platforms. Another, somewhat different approach, is called Plataforma.NET[3] and aims to be a hardware implementation of the CLR, so that CIL code can be run natively.

[1] Only non-commercial purposes are allowed.

[2] `http://www.go-mono.com/`

[3] `http://personals.ac.upc.edu/enric/PFC/Plataforma.NET/p.net.html`

3.3.1 Java VM vs .NET CLR

There are many similarities between Java and .NET technology. This is not strange, because both products serve the same market.

Both Java and .NET are based on a runtime environment and an extensive development framework. These development frameworks provide largely the same functionality for both Java and .NET. The most obvious difference between them is lack of language independence in Java. While Java's strategy is 'One language for all platforms' the .NET philosophy is 'All languages on one platform'. However these philosophies are not as strict as they seem. As noted in section 3.5 there is no technical obstacle for other platforms to implement the .NET Framework. There are compilers for non-Java languages like Jython (Python) [25] and WebADA [1] available for the JVM. Thus, the JVM in its current state, has difficulties supporting such a vast array of languages as the CLR. However, the multiple language support in .NET is not optimal and has been the target of some criticism.

Although the JVM and the CLR provide the same basic features they differ in some ways. While both CLR and the modern JVM use JIT (Just In Time) compilation the CLR can directly access native functions. This means that with the JVM an indirect mapping is needed to interface directly with the operating system.

3.4 Common Language Infrastructure

The entire CLI has been documented, standardized and approved [23] by the European association for standardizing information and communication systems, Ecma International[1]. Benefits of this CLI for developers and end-users are:

- Most high level programming languages can easily be mapped onto the Common Type System (CTS);
- The same application will run on different CLI implementations;
- Cross-programming language integration, if the code strictly conforms to the Common Language Specification (CLS);
- Different CLI implementations can communicate with each other, providing applications with easy cross-platform communication means.

This interoperability and portability is, for instance, achieved by using a standardized meta data and intermediate language (CIL) scheme as the storage and distribution format for applications. In other words, (almost) any programming language can be mapped to CIL, which in turn can be mapped to any native machine language.

The Common Language Specification is a subset of the Common Type System, and defines the basic set of language features that all .NET languages should adhere to. In this way, the CLS helps to enhance and ensure language interoperability by defining a set of features that are available in a wide variety of languages. The CLS was designed to include all the language constructs that are commonly needed by developers (e.g., naming conventions, common primitive types), but no more than most languages are able to support [33]. Figure 3.2 shows the

[1] An European industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) Systems. Their website can be found at `http://www.ecma-international.org/`.

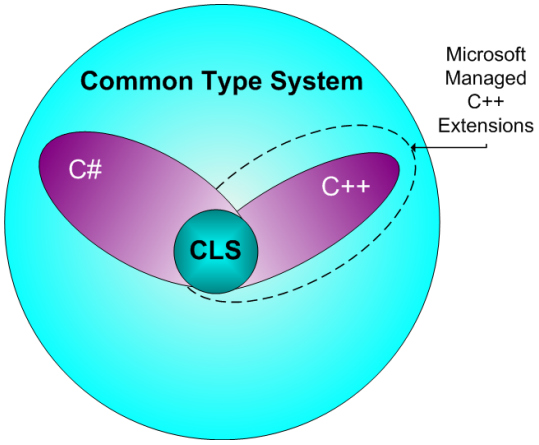


Figure 3.2: Relationships in the CTS

relationships between the CTS, the CLS, and the types available in C++ and C#. In this way the standardized CLI provides, in theory[1], a true cross-language and cross-platform development and runtime environment.

To attract a large number of developers for the .NET Framework, Microsoft has released CIL compilers for C++, C#, J#, and VB.NET. In addition, third-party vendors and open-source projects also released compilers targeting the .NET Framework, such as Delphi.NET, Perl.NET, IronPython, and Eiffel.NET. These programming languages cover a wide-range of different programming paradigms, such as classic imperative, object-oriented, scripting, and declarative languages. This wide coverage demonstrates the power of the standardized CLI.

Figure 3.3 shows the relationships between all the main components of the CLI. The top of the figure shows the different programming languages with compiler support for the CLI. Because the compiled code is stored and distributed in the Common Intermediate Language format, the code can run on any CLR. For cross-language usage this code has to comply with the CLS. Any application can use the class library (the FCL) for common and specialized programming tasks.

3.5 Framework Class Library

The .NET Framework class library is a comprehensive collection of object-oriented reusable types for the CLR. This library is the foundation on which all the .NET applications are built. It is object oriented and provides integration of third-party components with the classes in the .NET Framework. Developers can use components provided by the .NET Framework, other developers and their own components. A wide range of common programming tasks (e.g., string management, data collection, reflection, graphics, database connectivity or file access) can be accomplished easily by using the class library. Also a great number of specialized development tasks are extensively supported, like:

- Console applications;

[1]Unfortunately Microsoft did not submit all the framework classes for approval and at the time of writing only the .NET Framework implementation is stable.

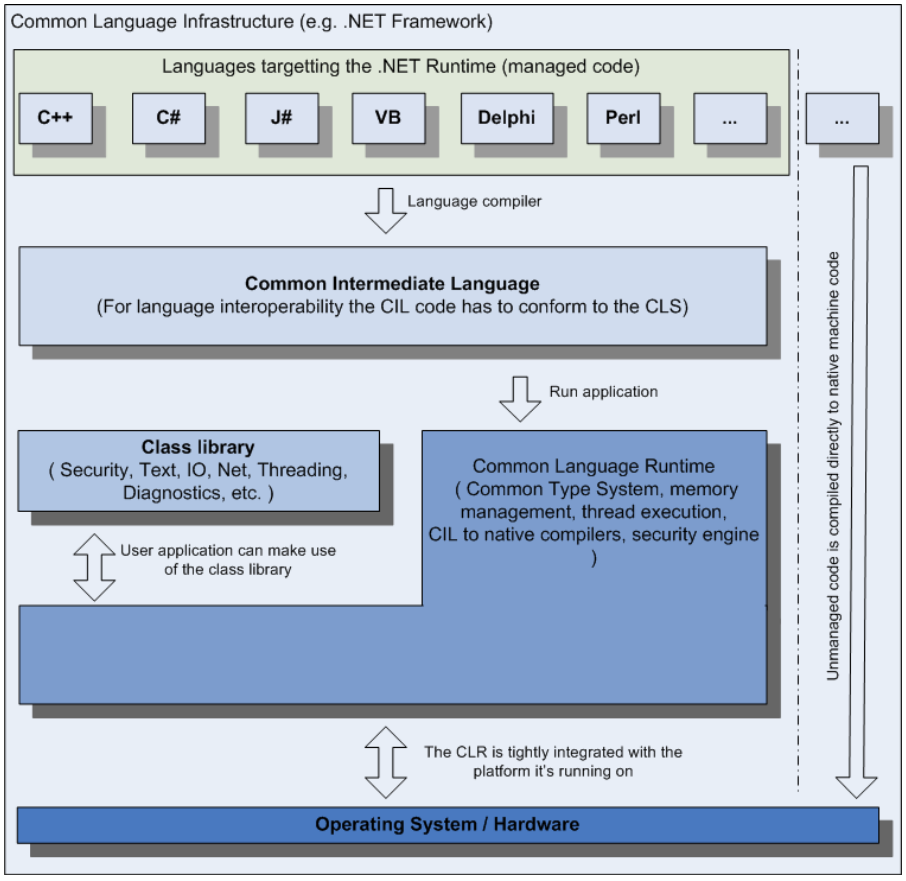


Figure 3.3: Main components of the CLI and their relationships. The right hand side of the figure shows the difference between managed code and unmanaged code.

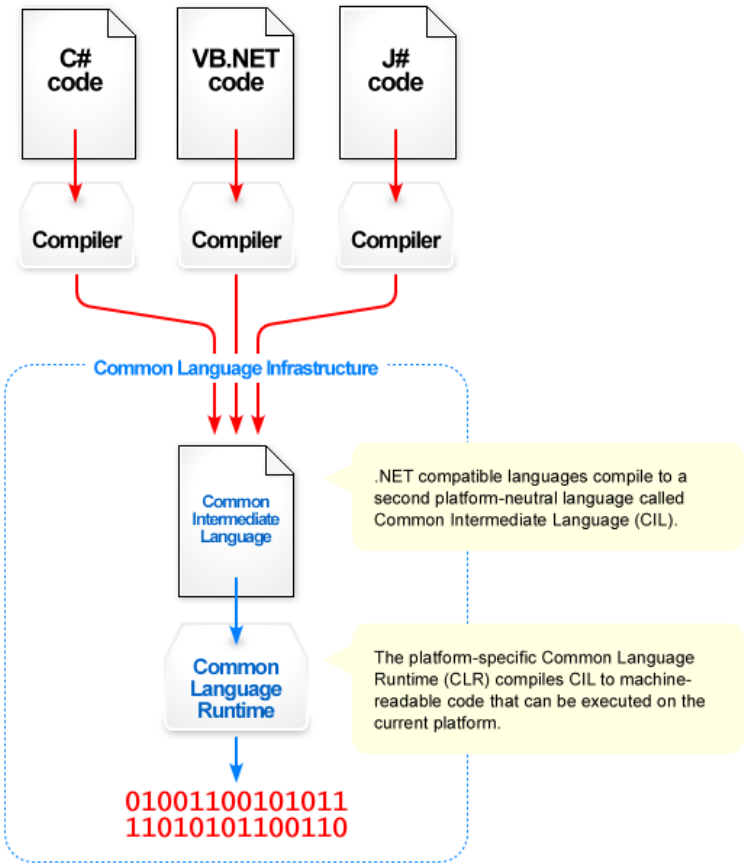


Figure 3.4: From source code to machine code

- Windows GUI applications (Windows Forms);
- Web applications (Web Forms);
- XML Web services;
- Windows services.

All the types in this framework are CLS compliant and can therefore be used from any programming language whose compiler conforms to the Common Language Specification (CLS).

3.6 Common Intermediate Language

The Common Intermediate Language (CIL) has already been mentioned briefly in the sections before, but this section will describe the IL in more detail. All the languages targeting the .NET Framework compile to this CIL (see Figure 3.4).

A .NET compiler generates a *managed module* which is an executable designed to be run by the CLR [47]. There are four main elements inside a managed module:

- A Windows Portable Executable (PE) file header;
- A CLR header containing important information about the module, such as the location of its CIL and metadata;

- Metadata describing everything inside the module and its external dependencies;
- The CIL instructions generated from the source code.

The Portable Executable file header allows the user to start the executable. This small piece of code will initiate the just-in-time compiler which compiles the CIL instructions to native code when needed, while using the metadata for extra information about the program. This native code is machine dependent while the original IL code is still machine independent. This way the same IL code can be JIT-compiled and executed on any supported architecture. The CLR cannot use the managed module directly but needs an assembly.

An assembly is the fundamental unit of security, versioning, and deployment in the .NET Framework and is a collection of one or more files grouped together to form a logical unit [47]. Besides managed modules inside an assembly, it is also possible to include resources like images or text. A manifest file is contained in the assembly describing not only the name, culture and version of the assembly but also the references to other files in the assembly and security requests.

The CIL is an object oriented assembly language with around 100 different instructions called OpCodes. It is stack-based, meaning objects are placed on an evaluation stack before the execution of an operation, and when applicable, the result can be found on the stack after the operation. For instance, when adding two numbers, first those numbers have to be placed onto the stack, second the add operation is called and finally the result can be retrieved from the stack.

```
1  .assembly AddExample {}
2
3  .method static public void main() il managed
4  {
5    .entrypoint            // entry point of the application
6    .maxstack 2
7
8    ldc.i4 3               // Place a 32-bit (i4) 3 onto the stack
9    ldc.i4 7               // Place a 32-bit (i4) 7 onto the stack
10
11   add                    // Add the two and
12                          // leave the sum on the stack
13
14   // Call static System.Console.Writeline function
15   // (function pops integer from the stack)
16   call void [mscorlib]System.Console::WriteLine(int32)
17
18   ret
19 }
```

Listing 3.1: Adding example in IL code

To illustrate how to create a .NET program in IL code we use the previous example of adding two numbers and show the result. In Listing 3.1 a new assembly is created with the name `AddExample`. In this assembly a function `main` is declared as the starting point (**`entrypoint`**) of this assembly. The **`maxstack`** command indicates there can be a maximum of two objects on the stack and this is enough for the example method. Next, the values 3 and 7 are placed onto the stack. The `add` operation is called and the results stays on the stack. The method `WriteLine` from the .NET Framework Class Library is called. This method resides inside the `Console` class placed in the `System` assembly. It expects one parameter with a **`int32`** as its type that will be retrieved

from the stack. The `call` operation will transfer the control flow to this method passing along the parameters as objects on the stack. The `WriteLine` method does not return a value. The `ret` operation returns the control flow from the main method to the calling method, in this case the runtime. This will exit the program.

To be able to run this example, we need to compile the IL code to bytecode where each OpCode is represented as one byte. To compile this example, save it as a text file and run the *ILASM* compiler with as parameter the filename. This will produce an executable runnable on all the platforms where the .NET Framework is installed.

This example was written directly in IL code, but we could have used a higher level language such as C# or VB.NET. For instance, the same example in C# code is shown in Listing 3.2 and the VB.NET version is listed in Listing 3.3. When this code is compiled to IL, it will look like the code in Listing 3.1.

```
1 public static void main()
2 {
3       Console.WriteLine((int) (3 + 7));
4 }
```

Listing 3.2: Adding example in the C# language

```
1 Public Shared Sub main()
2       Console.WriteLine(CType((3 + 7), Integer))
3 End Sub
```

Listing 3.3: Adding example in the VB.NET language

CHAPTER 4

Problem statement

The Composition Filters approach uses a concern specification to define the aspects imposed on the base system (see section 1.3.1). Separating aspect and component parts provides a way to use one aspect language that can target many component languages, as long as this aspect language is based on the interface specification of the component part. To demonstrate this separation the Compose* project aims at providing a component language (or programming language) independent implementation of the Composition Filters model. In other words, the Compose* project should make no assumptions about the component language used for defining the implementation parts. The .NET architecture looks very suitable for achieving this aim, since in principle it can support a wide range of different component languages, with a simple shared object model.
Because of the asymmetric composition approach used by Composition Filters, somewhere in the compilation, load, or run phase of an application the component and aspect languages have to be coupled. This process is called *weaving*. To enforce filter execution at run-time three different approaches can be followed (see section 1.3.2 for background information on aspect weaving): source code weaving, intermediate language weaving, and adapting the virtual machine.
The first approach, source code weaving, can be implemented as a source code weaver, i.e. the original source code and the aspect code are automatically combined by a tool, this output is given to the native source language compiler for compilation. For example SourceWeave.NET [24] attempts to overcome the source language dependency, usually associated with source code weaving, by using the .NET CodeDOM model. Unfortunately the .NET CodeDOM model is a standard representation for languages conforming to the Common Language Specifications (CLS) only. Even when interoperability with components written in other languages is not a design issue, the use of the .NET CodeDOM model by SourceWeave.NET still places restrictions on language constructs which can be used in the source code. The use of CSharp nested namespaces is for example not mappable to the .NET CodeDOM model, the nested namespace hierarchy will be flattened out. Furthermore, a .NET CodeDOM parser is still required for every language, and only a small portion of the languages targeting the .NET Framework provide such a parser today. Of course source code weaving also can rely on the application

developer instead of a tool. The application developer has to include the necessary code to link to a run-time execution engine for the filters (e.g. associating aspects via attributes [51] or objects extending a specialized message filtering object [14]). Two major disadvantages of this approach are that the source code is always needed and extending an existing application with Composition Filters can become a difficult task, as this approach can put restrictions on the design of the application. For example the approach of extending a specialized message filtering object requires that the object in the existing application does not already extend another object, since multi-inheritance is not possible in most languages.
The second approach, intermediate language weaving, relies on an aspect weaver to modify already compiled code, i.e. the aspect weaver uses the output of the native source language compiler. In the .NET architecture (see section 3.2 for information about the .NET architecture) all supported programming languages are compiled to the *Common Intermediate Language (CIL)*. This means an aspect weaver for the .NET architecture has to modify programs expressed in the CIL. Two advantages over the first approach are that the source code is no longer needed and the addition of filters does not restrict the design of the base system. in any way.
The third and last approach consists of adapting the virtual machine. In the .NET architecture this means adapting the Common Language Runtime (CLR). In addition to the disadvantages of this approach mentioned in section 1.3.2, the source code of the CLR of the .NET Framework is not publicly available making the adaptation of that CLR not an option. Only the public *Common Language Infrastructure (CLI)* projects Mono ([41]) and ROTOR ([48]) have the code of their CLR available so that it is possible to modify it to support weaving. Due to the '*under development*' status of these projects and the effort involved to keep an up to date modified CLR adapting these CLRs is currently not a viable option.

In a wider perspective, Composition Filters can be seen as just one of the many AOP solutions around today. AOP solutions that aim at weaving the aspects into code can embrace the CIL, as abstracting from source code knowledge is already done by native compilers. The CIL code has to be transformed to include the behavior defined in aspects. But in doing so the AOP language has the potency to bring AOP to all programming languages compiled to the CIL. As transforming the CIL code may not be as simple as it sounds, the development of AOP tools targeting the .NET architecture can benefit from a tool doing the CIL code transformations for them.

The set of code transformations offered by such a tool should be expressive enough to support at least the most basic set of AOP constructs. Apart from expressiveness, the way code transformations are defined for a certain application has to be in a human readable format. These transformation definitions have to be created by the developers of the AOP tools and therefore the definition scheme should not require a lot of effort to understand. Of course the usage of such a transformation tool is not bound to AOP tools alone. Every project requiring transformations of the CIL code provided by this tool can make use of it.

To overcome the limitations of source code weaving and provide a general way for AOP solutions to weave aspects this thesis will focus on the development of an usuable aspect weaver for the CIL. Hence a proper set of aspect-oriented crosscutting locations is defined together with the possible crosscutting behavior at these locations. In the next step a mapping from these crosscutting locations to well-defined points in the CIL will be made. Based on this information a transformation tool can be designed and implemented to perform the necessary transformations on the CIL code. As providing the Compose* project with a way to enforce filter execution at run-time is one of the goals, the created transformation tool should be embedded in the Compose* project. This can be done by providing a mapping from the Composi-

tion Filters model to the transformation definitions used by the tool. In terms of the Compose* architecture, as described in section 2.4, this means that the compile time layer has to create the transformation definitions and execute the transformation tool.

CHAPTER 5

Understanding the Common Intermediate Language

In section 3.2 a general overview of the .NET architecture was given. In this chapter we will discuss in more detail the Common Intermediate Language (CIL), the intermediate language used by the .NET platform, and the way the CIL is deployed.

Section 5.1 describes the contents of an assembly, the unit of deployment for the .NET platform, and how the CIL is embedded within. The execution of CIL code is performed by the stack-based Common Language Runtime (CLR). Section 5.2 will explain the two mechanisms used to store state information within the CLR. In section 5.3 the Common Type System (CTS) is discussed in some detail, to give the reader a better understanding of the type system (e.g. primitive types like int or float, classes, or interfaces) used by the .NET platform. Following the CTS we will discuss briefly the instruction set used by the CLR in section 5.4. The instruction set consists of base instructions (e.g. addition, or subtraction), and object model instructions (e.g. object initialisation, or method calls). Finally section 5.5 gives an example of a simple *HelloWorld* program expressed in the CIL to show some of the concepts described in this chapter.

5.1 The assembly, unit of deployment

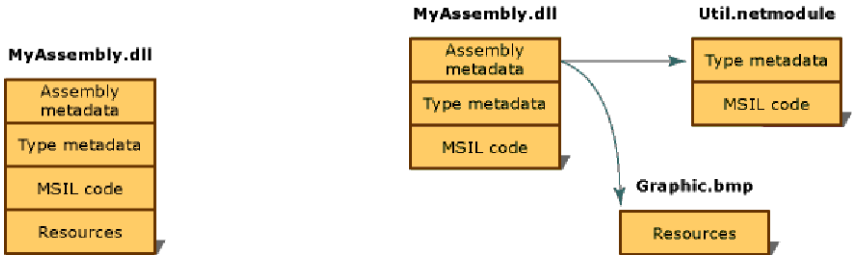
.NET application executables and dynamic link libraries are deployed through *assemblies*, these are the shared units within the runtime environment (the Common Language Runtime, or CLR for short). An assembly is a compiled and versioned collection of code and metadata that forms a single functional unit [39].

Each assembly contains a *manifest* (also called *assembly metadata*), which states the name of the assembly, the version, the locale, a list of files that form the assembly, any dependencies the assembly has, and which features are exported (type metadata) [36]. Optionally an assembly can include *.NET Framework types* (interfaces and classes) and *resources*. In a single assembly the interfaces and classes may be spread over multiple namespaces. They are stored in the intermediate language code, better known as the Common Intermediate Language (CIL), used by the CLR. Microsoft also uses the term 'Microsoft Intermediate Language' (MSIL) instead

of CIL for the intermediate language used by the .NET Framework. Resources embedded or referenced in an assembly can be files of any type, e.g. bitmaps, JPEG files, resource files, or XML files.

An assembly can be composed of multiple *modules*, or *netmodules* [39, section Building a Multifile Assembly]. These modules contain the .NET Framework types and can be compiled separately. The main reason to combine multiple modules in a single assembly is to combine modules written in different languages and deploy them as a single file.

Although an assembly can be thought of as a single logical unit it can consist of more than one physical file [1] [36]. Figure 5.1 shows both a single-file and a multifile assembly. In the single-file assembly *MyAssembly.dll* the manifest, type metadata, MSIL code and resources are contained in a single physical file. On the other hand in the multifile assembly a part of the type metadata and MSIL code has been moved to a different physical file, *Util.netmodule*. Since an assembly only can have one manifest, the *Util.netmodule* is compiled as a *module*. Also the resource *Graphic.bmp* is not embedded in the assembly file but rather referenced in the manifest.



Single-file assembly Multifile assembly

Figure 5.1: Single-file and multifile assembly layout.
Source: Assemblies [36, section Contents].

The assembly layout described above is that of a *static assembly* [36]. Static assemblies are stored on disk, as portable executable (PE) files (see section 5.1.1). They are the most commonly used type of assembly. However with the powerful reflection capabilities of the .NET Framework it is also possible to create assemblies at runtime. Assemblies created this way are called *dynamic assemblies* and run directly from memory.

5.1.1 Portable Executable files

PE files are the binary distribution format of 32-bit Windows (Win32) programs [44]. They contain a stub MSDOS program to make sure the program runs inside a compatible Windows environment. .NET binaries must contain a Win32 stub to use .NET to run the actual program, or to inform the user that .NET is required to run the program. The stubs together with the section table, of which the records point to all the different sections in the PE file, are called the *PE file header*. Following this PE file header a .NET PE file has three sections of data:

MSIL code: This is the section that actually gets compiled to native code and executed by the CLR. To optimize performance a method is compiled at the moment it is called, this is

[1] The files that make up a multifile assembly are not physically linked by the file system. Rather, they are linked through the assembly manifest and the CLR manages them as a unit [36].

called *Just-In-Time (JIT) compilation*. Of course this compiled version is buffered to serve subsequent calls to the same method without an additional compilation penalty.

Metadata: In the metadata section the content of the MSIL code is described. For example which methods are provided, which arguments the methods accept, and which type the methods return.

Manifest: The assembly manifest is important for the CLR to load other components the assembly requires in order to run. To find the correct referenced assembly, the CLR also uses the assembly manifest of the referenced assembly. It can compare the information provided in the reference with the information of an assembly.

5.2 The activation record and evaluation stack

The CLR makes use of two structures to store the execution state of a program: the *activation record* and the *evaluation stack*. The activation record holds all the activation data of a single execution of a method. To exchange data between successive operations the evaluation stack is used.

For each execution of a method a new activation record is created. It stores the arguments and the local variables of the method in two separate numbered collections. Of course there may be zero or more arguments, and zero or more local variables. The exact amount and types of both arguments and local variables form part of the definition (or signature) of the method and are hence known at the time the activation record is initialized [17]. The numbering of arguments and local variables is a logical numbering, i.e. an argument or a local variable is allocated an index based on its position in the declaration, regardless of its size.

CIL operations are performed on the evaluation stack and can be divided into three main categories [17]:

Load instructions: These operations *push* values onto the evaluation stack, making the stack deeper.

Value instructions: Operations performed on values already on the stack belong to this category, e.g. arithmetic instructions. Depending on the number of incoming and outgoing values of the instruction the evaluation stack becomes deeper or shallower.

Store instructions: These operations *pop* the top element of the stack and store the value in the specified place. The evaluation stack will become shallower through the use of these operations.

The evaluation stack can only be accessed from the top, i.e. there is no way to access any element other than the top element [17]. Also it is not possible to access the top element without removing it from the stack. Just like the activation record has logical numbered collections, the evaluation stack is a logical numbered stack. In other words the stack entries are type instances and only accessible as such, contrary to a stack which is accessed by bytes or words. This means the depth of the stack is the number of elements on the stack, regardless of their type.

5.3 The Common Type System

The basic CLR type system, or Common Type System (CTS), can logically be divided into two subsystems: the value types system, and the reference types system. Figure 5.2 depicts both subsystems of the CTS. A sequence of bits in memory can make up a value type, e.g. a 32-bit integer. Value types are considered equal if their sequence of bits is identical [61]. On the other hand reference types contain the memory address of the value, also known as its identity. Comparing reference types, therefore, can be done by identity or by equality. If two references refer to the same object (they refer to the same memory location) it means they have the same identity; if two references refer to two different objects that have the same sequence of bits, i.e. the same data, they are equal [61].

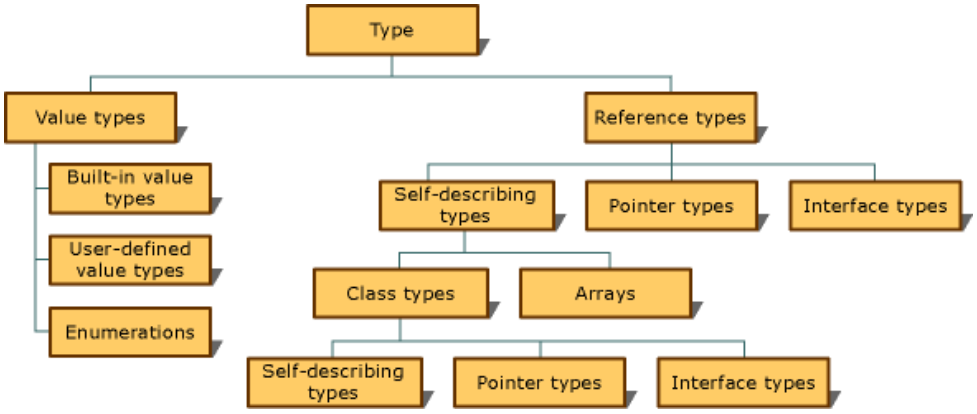


Figure 5.2: The Common Type System.
Source: Common Type System [38, section Overview].

5.3.1 Value types

Value types are used to represent the simple or primitive types of many languages, e.g. int or float. They can be used as method arguments, local variables, or return types of methods. A value type always directly inherits from System.ValueType or System.Enum (which in turn inherits from System.ValueType). Furthermore value types are sealed, which means other types can not inherit from them [61].

For every value type there exists a corresponding object type, known as its *boxed type* [61]. There are two operations that can be performed with types:

Boxing: The operation performed on a value type that copies the data from the value into an object of its boxed type allocated on the garbage collected heap.

Unboxing The operation performed on a boxed type that returns a pointer to the actual value, i.e. the sequence of bits in memory, held in a boxed object.

The fact that all value types can be converted to their corresponding object types allows all values in the type system to be treated as objects if required [61]. Note that this unifies the two different types in the CLR, because every type can be treated as a subtype of *System.Object* (see section 5.3.2 for more details).

Most languages do not allow developers to implement their own value types, but the CTS does. Of course the CTS has a basic set of value types already defined, the built-in value types [61].

5.3.1.1 Built-in value types

Table 5.1 lists all the built-in value types of the CTS in the categories *integer*, *floating point*, *logical* and *other* [2, 17, 61]. In the column "Type" the name of the type in the Base Framework is shown. The next column, "CLS?" states if the type is a Common Language Specification (CLS) type; only CLS types can be used in cross-language value exchange. The column "CIL name" shows the name of the type as used in the CIL. The "Suffix" column shows the suffix used to differentiate between CIL instruction targeting different types, e.g. *conv.i1* and *conv.i8* which, respectively, converts the top-of-stack element to a 1-byte integer and a 8-byte integer. Finally in the column "Description" a short description of the type is given.

Category	Type	CLS?	CIL Name	Suffix	Description
Integer	SByte	×	int8	i1	8-bit signed integer
	Int16	√	int16	i2	16-bit signed integer
	Int32	√	int32	i4	32-bit signed integer
	Int64	√	int64	i8	64-bit signed integer
	Byte	√	unsigned int 8	u1	8-bit unsigned integer
	UInt16	×	unsigned int 16	u2	16-bit unsigned integer
	UInt32	×	unsigned int 32	u4	32-bit unsigned integer
	UInt64	×	unsigned int 64	u8	64-bit unsigned integer
Floating point	Single	√	float32	r4	IEEE 32-bit floating point type
	Double	√	float64	r8	IEEE 64-bit floating point type
Logical	Boolean	√	bool	u1	Boolean type
Other	Char	√	wchar	u2	a Unicode (16-bit) character
	Decimal	√	decimal		a 96-bit decimal value
	IntPtr	√	native int		a signed integer, the size of which depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform)
	UIntPtr	×	unsigned native int		an unsigned integer, the size of which depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform)
Class types	Object	√	object		base class for all class types
	String	√	string		an immutable, fixed-length string of Unicode characters

Table 5.1: Built-in value and reference types.

5.3.2 Reference types

In contrast to value types, the reference types do not inherit from System.ValueType or System.Enum. These reference types can inherit from any other class though. Because reference

types are always allocated on the garbage collected heap and the garbage collector is free to move objects during execution, they are accessed via strongly typed references[1] rather than directly. When the garbage collector moves an object these strongly typed references are updated as part of the relocation process [61].
In the CTS three categories of reference types can be identified (see figure 5.2): self describing types (class types and array types), pointer types, and interface types.

The *self describing types* can be split into *class types* and *array types*. The class types are the set of all exact types for all objects. In many object-oriented languages class types are simply referred to as 'classes'. An instance of a class type is called an 'object'. The built-in class types are described later in this section. Array types are defined by specifying the element type, the number of dimensions (rank), and the upper and lower bounds of each dimension [37]. Exact array types (fixed number of dimensions and bounds) are automatically created by the CLR when needed [38, section Arrays]. This means the CLR will handle dynamically supplied number of dimensions and bounds without the need for any special constructs.

Pointer types provide the CLR with a way of specifying the location of either code or a value. Three pointer types can be identified [61]:

Managed pointers: These pointers are known to the garbage collector and will be updated if they refer to an item that is moved by the garbage collector. Managed pointers are CLS compliant.

Unmanaged function pointers: They refer to the address of the function and are similar to function pointers in C++.

Unmanaged pointers: They are similar to unmanaged function pointers but refer to values instead of function addresses. Unmanaged pointers are not CLS compliant, many languages have no syntax to define or use them.

Interface types are partial specifications of types to share a common contract between types. Implementers are bound by this contract to provide implementations of the interface members. Object types may support many interface types and an interface type may inherit from other interface types. Members that can be defined in an interface type are: methods (static and instance), fields (static), properties, and events [61].
There are no built-in interface types included in the CLR, although a number of interface types are provided by the Base Framework.

5.3.2.1 Built-in reference types

The CTS has a set of already defined reference types in the category class types: *Object* and *String* [61].
System.Object or simply Object is the CLR type from which all class types inherit, either directly or indirectly. The most important result of this is the enforcement of a singular rooted inheritance hierarchy for all CLR types. Even value types can be treated as subtypes of Object through *boxing*.
The other built-in reference type is System.String or simply String. This class is sealed and

[1] Strongly-typed languages, like the CIL, have a strict enforcement of type rules. Many strongly-typed languages also provide strongly typed references to objects, i.e. the pointer knows the type of the object it is referring to.

immutable, i.e. respectively no type can be subtyped from it and a method modifying a string creates a new string. Both facts allow for a very efficient implementation in the CLR [61]. For instance, strings are always safe for multithreading, since there's nothing a thread can do that would mess up another thread by modifying a string, since strings cannot be modified.
Details of the built-in reference types *Object* and *String* can be found in table 5.1 in the category class types [2, 61].

5.4 The CIL instruction set

The CIL instruction set contains about 220 instructions, also called opcodes. Roughly two thirds are base instructions, e.g. addition, and subtraction. The other instructions serve the support of the object model [17]. With only 220 instructions one byte (256 different states) is enough to store the exact instruction. A CIL instruction can be followed by one argument, in contrast to Java byte-code instructions which can be followed by more than one argument [29]. By definition the instruction itself indicates whether or not it is followed by an argument, and the form of the argument (if any) it takes.

For optimalisation reasons a lot of instructions has so called *short forms* [17]. These short forms have the argument embedded into the instruction. Compare the following instructions for loading the arguments of a method: *ldarg <index>*, *ldarg.0*, *ldarg.1*, *ldarg.2*, and *ldarg.3*. The first instruction is the normal instruction to load the method arguments, and it takes the index of the method argument to load as instruction argument. The other four instructions are the short forms for loading the first four method arguments. So when loading one of the first four method arguments it is possible to save one byte, no index is needed, per load instruction by using the short form. Apart from the size optimalisation, the use of short forms also eliminates the reading of an instruction argument, the index in the example above, by the CLR.

To get a basic understanding of the different instructions the following sections will briefly describe them, categorized by their functionality. A complete list of instructions can be found in appendix A.

5.4.1 Load and store instructions

The load instructions, i.e. instructions to push values or addresses on the stack, can further be divided in three subcategories: loading values, loading constants, and loading addresses [17].

Values that can be pushed on the stack are static field values, instance field values, local variable values, method argument values, and array element values. The argument of these instructions specifies the details of the instruction.

Numbers, strings, and *null* are the constants that can be pushed on the stack. Loading numbers is done by four different instructions denoting their type: *ldc.i4* (int32), *ldc.i8* (int64), *ldc.r4* (float32), and *ldc.r8* (float64). In section 5.3.1.1 also the built-in value types boolean, int8, and int16 are shown. They do not have separate loading instructions, but rather are loaded as int32 values and converted to the right type. For example loading the boolean value *true* on the stack is done by loading the int32 value *1* and casting it to the *System.Boolean* type. Just like the short forms for loading values, there are ten short forms for loading the most used int32 values (-1 to 8).

Addresses that can be pushed on the stack, for example for passing method arguments by reference (*byref*), are the addresses of the values that can be pushed on the stack.

To store a top-of-stack value one can use the counterparts of the load instructions for values [17]. To store the value of a top-of stack address one can use the indirect store instructions. Since the store instructions on the stack are implicitly subject to the usual unary conversion rules[1] it's not necessary that the top-of-stack type is the same as the destination type. In case the destination type is *smaller* than the type on the stack, truncation will occur. E.g. it is valid to store an int32 to a int16 location, automatic truncation will occur. Note that automatic truncation will occur without any kind of warning, resulting in possible dataloss. To trap a possible overflow the conversion from int32 to int16 has to be made explicitly with a conversion instruction, for details see the next section.

5.4.2 Operate instructions

These instructions operate on values on the top of the stack and hence do not posses any instruction arguments. Since their operands are implicitly taken from the stack most of these instructions are polymorphic, i.e. the actual operator is chosen by the CLR to match the operand type(s) on the stack [17].

The operate instructions can be categorized in three groups [17]:

Arithmetic: The arithmetic instructions include addition, subtraction, multiplication, etc..

Logical: Logical instructions comprise the boolean comparisons, e.g. *and*, *or*, *not*, etc., and the shift instructions.

Type conversion: To convert the various numeric types the type conversion instructions can be used. There are two sets of type conversion instructions: non overflow-checked and overflow-checked instructions.

5.4.3 Branching and jumping instructions

Branching and jumping refer to continue program execution at an instruction other than the sequentially next one. Conditional branching allows a program to take two different paths depending on a condition. Jumping or unconditional branching on the other hand allow only one path to be taken. From a high-level language standpoint, branching and jumping are used for if statements and loops (e.g. a while loop). Note that this means that CIL instructions do not map uniquely to high-level language constructs, making identification of high-level language contructs in CIL code difficult, if not impossible. There are several branching instructions in the CIL which can be divided into three groups: conditional branch instructions, unconditional branch instructions, and the table switch instruction.

Both conditional and unconditional branch instructions take a label as instruction argument. In textual CIL, as described in section 5.5, a label is a distinct identifier followed by a colon [17]. A label has to be unique, only inside a method, logically resulting from the fact that the method boundary is also the branching boundary. In other words it is not possible to branch outside

[1] Implicit unary conversion is performed by the CLR with the use of the usual unary operators. A unary operator acts on one operand.

the scope of a method. But in binary CIL, i.e. the format used in assemblies and by the CLR, labels are defined as a positive or negative offset in bytes from the branch instruction. Note that the offset is in bytes and therefore the exact size of all instructions and their arguments have to be computed at compile time to get the right offset. For this reason all branch instructions also have a short form. The short form only takes one byte as its argument, in contrast to the normal form which takes 4 bytes, and is used to optimize jumps within 128 bytes of the branch instruction.

Conditional branch instructions take their operands from the stack and based on the boolean evaluation a branch occurs. For example the 'branch if equal' instruction *beq* compares the two elements on top of the stack for equality and only branches to the defined label if the two elements are equal. If the two elements are not equal, execution continues at the instruction directly following the branch instruction.

In fact there is only one unconditional branch instruction, *br*. In contrast to the conditional branch instructions no evaluation is performed, but a direct branch to the specified label occurs. Again the normal form takes a four byte argument and the short form takes a single byte argument.

For *switch* and *case* statements an indexed, indirect branch instruction exists, named *switch*. In listing 5.1 a typical example of the switch instruction in textual CIL is given.

Listing 5.1: Example of a typical format of the switch instruction [17, pg. 39]

```
1 <instructions to push selector value on the stack>
2 switch (  // start label table
3   lb01, // first label
4   ...,
5   lb07  // last label
6 )
7 br  lb08  // branch to default label
```

5.4.4 Miscellaneous instructions

Instructions that do not fit in the former three categories include calls, returns, and some non verifiable instructions operating on untyped data.

There are a few different call instructions for calling static and instance methods [17]. The four most important instructions are for: normal calls, virtual calls (instance only), calls by function pointer, and virtual calls by function pointer (instance only). Each call instruction has as instruction argument the full signature of the method to be called; the method arguments are taken from the top of the stack. In case of an instance call also the instance to make the call to should be pushed onto the stack, in front of the method arguments. The virtual call instructions allow calling inherited methods on an instance, i.e. calling a method defined in a parent type of the instance the call is made to.

There is only a single return instruction to return from a method, which returns from both *void* and value-returning functions. In the last case the return value should be left on the stack.

To efficiently implement some ANSI C functions, like *memset* and *memcpy*, the CIL contains a few non verifiable instructions [17]. These instructions will not be discussed here any further.

5.5 Example: A simple program written in the CIL

The CIL code used in assemblies and by the CLR is a binary code. But since binary code is very hard to read for humans a textual representatation of the CIL code exists. To make the concepts described in the previous sections more clear we will briefly discuss the textual representatation of a simple *HelloWorld* program written in the CIL. The complete CIL code can be found in appendix B.

Textual CIL begins with referencing the external assemblies it requires using the directive *.assembly extern*. The *HelloWorld* example requires the external assembly 'mscorlib', the core library of the .NET Framework. Listing 5.2 gives the CIL code referencing the core library 'mscorlib'. Apart from the name of the referenced assembly the public key token, if any, and the version number are stored. Both key token and version number are used to load the correct external assembly. The key token should prevent the loading of modified assemblies (signed assemblies are)

Listing 5.2: An external assembly reference in the CIL

```
1 .assembly extern mscorlib
2 {
3   .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
4   .ver 1:0:5000:0
5 }
```

Next to the external assembly reference(s) the information of the current assembly is given using the directive *.assembly* (see listing 5.3, lines 1-5). Among others the public key token, if any, and the version of the assembly is stored. Following this public information the private assembly information is given, like the module(s) making up the assembly (*.module*. Both public and private assembly information are given in listing 5.3. In the example only the module 'HelloWorld.exe' is embedded in the assembly (line 7).
Together all this information forms part of the assembly manifest, as described in section 5.1.

Listing 5.3: The assembly information in the CIL

```
1 .assembly HelloWorld
2 {
3  .hash algorithm 0x00008004
4  .ver 1:0:1720:26694
5 }
6
7 .module HelloWorld.exe
```

Next the CIL code states the namespace(s), *.namespace*, and class(es), *.class*, implemented by the assembly. The reason this information is explicitly printed out in the CIL code is the close relation with the binary CIL code and the metadata. In fact we are still looking at the representation of the metadata at this point. Listing 5.4 gives the structure of classes for the HelloWorld example.

Listing 5.4: The structure of classes in the CIL

```
1 .namespace HelloWorldExample
2 {
3   .class private HelloWorldMain
4     extends [mscorlib]System.Object
5   {
6   } // end of class HelloWorldMain
7 }
```

Finally the implementation of the classes is given, see listing 5.5. For each namespace and class the structure is repeated and extended with fields, *.field*, and methods, *.method*, to supply the implementation.
As can be seen in listing 5.5 at line 3 a class definition contains not only the access level (public, family, private) and the name of the class but also a series of other parameters. The complete list of possible parameters will not be discussed here, for now it is enough to keep in mind they exist. As can be seen on line 4 the 'HelloWorldMain' class extends the System.Object type, in fact this line is not even nessecary since all classes inherit by definition from System.Object (as described in section 5.3.2).

Listing 5.5: The implementation of classes in the CIL

```
1  .namespace HelloWorldExample
2  {
3    .class private auto ansi beforefieldinit HelloWorldMain
4      extends [mscorlib]System.Object
5    {
6      .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
7      {
8        .maxstack 1
9        IL_0000:  ldarg.0
10       IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
11       IL_0006:  ret
12     } // end of method .ctor
13
14     .method private hidebysig instance void Run() cil managed
15     {
16       .maxstack 2
17       IL_0000:  ldarg.0
18       IL_0001:  ldstr      "Hello world"
19       IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
20       IL_000b:  ret
21     } // end of method Run
22
23     .method private hidebysig static void Main(string[] args) cil managed
24     {
25       .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = (01 00 00 00)
26       .entrypoint
27       .maxstack 1
28       .locals init (
29         [0]class HelloWorldExample.HelloWorldMain
30       )
31       IL_0000:  newobj     instance void PeWeaverTests.HelloWorld::.ctor()
32       IL_0005:  stloc.0
33       IL_0006:  ldloc.0
34       IL_0007:  callvirt   instance void PeWeaverTests.HelloWorld::Run()
35       IL_000c:  ret
36     } // end of method Main
37
38   } // end of class HelloWorldMain
39 }
```

The *HelloWorld* example consists of three methods: *.ctor* (lines 6-12), *Run* (lines 14-21), and *Main* (lines 23-36). The method definition also contains more identifiers than its access level and its name. For example the methods *.ctor* and *Run* use the identifier *instance*, while the method *Main* uses the identifier *static*. These two identifiers are each others opposite; instance methods can only be called on an object, static methods can be called without an object.

In the implementation of the three methods one can see the labeling and CIL instructions with their arguments as described in section 5.4. The numbering of the labels is based on the size in bytes of the instructions and their arguments. For example the *call* instruction itself is one byte (as all instructions are one byte) and takes a four byte instruction argument, so in total it takes

five bytes which is reflected in the labeling.
In the *Run* method a message (in the form of a string) is printed to the screen (line 19). The called method *WriteLine* takes one parameter of type string. This parameter has to be on the stack before the call instruction is processed. Therefore the call instruction is proceeded by the *ldstr* instruction (line 18) which pushes the string 'Hello world' onto the stack.
The *Main* method shows a few optional method directives, like *entrypoint* (line 26, which denotes that this method is to be called by the CLR when the class is loaded) and *locals* (line 28, which lists the local method variables).

5.6 Summary

In this chapter the CIL, the Common Intermediate Language, and the way it is deployed by assemblies were introduced. The assembly is the shared unit within the Common Language Runtime (CLR) and has a layout containing the manifest, the modules, the code, and the resources.
To get a basic understanding of the way the CIL operates the activation record, evaluation stack, the Common Type System (CTS), and the CIL instruction set were discussed. Both the activation record and the evaluation stack exist due to the fact that the CLR is a stack-based virtual machine. They are used to store values passed from one method to another respectively to store values passed from one instruction to the next. The type system used by the CLR, the CTS, is a singular rooted inheritance hierarchy with the reference type *System.Object* at the top. Apart from reference types the CTS consists of value types, which can be mapped to reference types by the boxing and unboxing operations. Roughly 220 instructions form the CIL instruction set, this instruction set consists of both base instructions and instructions to support the object model.
Finally the textual CIL code of a simple program was discussed to show the different concepts of the CIL in practice.

CHAPTER 6

Mapping AOP constructs to the Common Intermediate Language

The AOP paradigm, introduced in chapter 1, solves the problem of crosscutting concerns by defining the *aspect* to capture the location and behavior of crosscutting concerns. The CIL is a language with support for objects but not for aspects as we have seen in the previous chapter. To add support for aspects to the CIL we have to map the location of crosscutting concerns (in other words the crosscutting locations) to a well-defined point in the CIL. At this well-defined point in the CIL the behavior of the aspect can be added. In this chapter we will make an inventory of crosscutting locations and how they can be mapped to well-defined points in the CIL.

Section 6.1 gives an overview of common crosscutting locations which can be identified in many AOP solutions. In section 6.1.3 the different possibilities for the concern implementation are briefly explained. Which points in the CIL we can see as well-defined points is investigated in section 6.2. Once we have identified the crosscutting locations and the well-defined points in the CIL we can create a mapping from crosscutting locations to well-defined points in the CIL, this is done in section 6.3. Section 6.4 will narrow down the mapping list of section 6.3 to the list of well-defined points in the CIL which have to be supported by our tool. Finally section 6.5 will wrap up this chapter.

6.1 Crosscutting locations

As described in section 1.4.1 there are two kinds of crosscutting mechanisms: *dynamic crosscutting* and *static crosscutting*. All crosscutting locations can be catagorized in thse two crosscutting mechanisms.

A crosscutting location is a well-defined point in the execution (dynamic crosscutting location) or in the type structure (static crosscutting location) of a program. A crosscutting concern must be attached to a crosscutting location to have any usefulness in a program. Many crosscutting locations can be defined as a combination of other crosscutting locations. The minimal set of crosscutting locations in which all crosscutting locations can be expressed is the set of *primitive*

crosscutting locations.

These primitive crosscutting locations form the basis for the mapping from AOP constructs to weave points in the CIL. The sections 6.1.1 and 6.1.2 will describe in more detail the primitive dynamic crosscutting locations respectively the primitive static crosscutting locations.

6.1.1 Dynamic crosscutting locations

The following primitive dynamic crosscutting locations have been identified and will be described here briefly:

Application start: This is the location where the first user code is executed after the application has started, i.e. after default application initialization code. For AOP implementations based on an interpreter this is the location to load all required information for the interpreter.

Constructor call: The constructor call location is defined as the location where an initial constructor of the object is called, i.e. not for 'super' or 'this' constructor calls.

Object pre-initialization: The pre-initialization of an object is the part from entering the first-called constructor to the call of the super constructor. The location is identified by a certain signature.

Object initialization/constructor execution: At this location the code of the constructor is executed, after its 'super' or 'this' constructor call.

Method call: This is the location where a call to a method is made. Note that the execution context is at the side of the caller.

Method execution: This is the location where the code of a method is executed, i.e. the execution context is at the side of the callee (the executed method).

Field access (get and set): There are two forms of field access, namely getting the value of a field and setting the value of a field. Both forms have their execution context at the side of the getter or setter, comparable with the method call context.

Exception handler: The location where an exception handler executes is identified with this crosscutting location. The exception being handled is available for inspection.

This: Every location when the currently executing object is an instance of a certain type.

Target: Every location when the target executing object is an instance of a certain type.

Arguments: Every location when the arguments are instances of certain types.

Control flow (if) every join point in the control flow of each join point P picked out by Pointcut, including P itself.

Assignment: Every location where a value gets assigned.

The crosscutting locations *object pre-initialization*, *construction exection*, *method call*, and *method execution* are not uniquely identified by the name of the constructor or method. In many languages a constructor or method can have different overloaded forms which take different arguments. These different overloaded forms are said to have different *signatures*. The four crosscutting locations named above are therefore identified by their full name and optionally by signature.

6.1.2 Static crosscutting locations

With static crosscutting, the program structure can be changed, for example methods and fields can be added to the type declaration. The use of aspects allows the change of multiple types with one single statement. In AspectJ the feature of static crosscutting is called *inter-type declarations*.

6.1.3 Concern implementation

When a crosscutting location is found, the concern implementation (the additional code to execute) can be inserted. We identify four insertion possibilities at a crosscutting location:

1. before the crosscutting location;
2. after the crosscutting location;
3. around the crosscutting location (combination of before and after);
4. replace the crosscutting location.

Insertion before the crosscutting location:
Inserting additional code to execute before the identified crosscutting location is one of the four concern implementations. The additional code has to be executed before the point identified with the crosscutting location is executed, e.g. before the call to a method is made.

Insertion after the crosscutting location:
The additional code to be executed is inserted after the identified crosscutting location. For example after the call to a method is completed. A special case can be added in the event the method returns an exception. That way the execution of additional code can be prevented if the method returned with an exception.

Insertion around the crosscutting location:
This is a combination of before and after insertion. The benefit of using around insertion instead of before and after separately is that you can re-use information in the after part, which was stored in the before part. In other words, a local executing contect can be carried from the before part to the after part. For example the value of a variable set in the before part can be accessed in the after part.

Replacement of the crosscutting location:
Certain crosscutting locations allow the complete replacement of the crosscutting location code

with new code. For example the crosscutting location *method call*: the call to the method can be replaced with a call to another method.

6.2 Weave points

Of course a list of wanted crosscutting locations can be made, but these crosscutting locations have to be identifiable in the program execution trace. Otherwise it is impossible to execute the concern implementation at the required point in the program execution. Reasoning about program execution in this context can be seen as the execution at runtime (dynamic weaving) or as an interpretation of the code (static weaving).
Since we are interested in static weaving we have to identify unique points in the CIL code of a program. We will call these points in the CIL code *weave points* from now. Chapter 5 introduced CIL code and gives an idea of what would be possible weave points.
As long as there is a one to one mapping from the high level language constructs (constructs in the program language used by the developer on which crosscutting locations are defined) to CIL code constructs (our weave points) no problems occur. But some high level language constructs like conditional expressions and loops are all mapped to the same branching CIL code constructs. This makes it very difficult or sometimes impossible to map a CIL code construct to a crosscutting location. Therefore these CIL code constructs can not be used as weave points. We can split weave points into structural and executional weave points, which will be explained in more detail in section 6.2.1 respectively section 6.2.2.

6.2.1 Structural weave points

Structural weave points are points in the CIL code where it is possible to add, change or delete the structure of a class. For example adding a new method to a class. The following list is an overview of all the structural weave points in the CIL we have identified.

Constructor: Constructors are special methods (methods are recognised by a *.method* keyword) identified by their name *'.cctor'* in CIL code.

Method: Methods are identified by the *.method* keyword.

Field: Fields are identified by the *.field* keyword in the variable declaration part of a method.

Attribute: Attributes are identified by the *.custom instance* keyword in the variable declaration part of a method.

Delegates: Delegates are a type-safe mechanism to implement function pointers. They are also used to implement *event types*. In the CIL code, delegates are identified by the fact that they have to extend the *System.MulticastDelegate* class.

Property: Properties are identified by the *.property* keyword.

Inheritance relationships: Depending on the actual inheritance relationship it is identified by the *extends* or *implements* keywords following the name of the class in the class definition (a class definition is recognised by the *.class* keyword).

6.2.2 Executional weave points

Executional weave points are points identified in the program flow and mostly found in the CIL instruction set itself.

Application start: The method defined as starting point for an application is recognised by the *.entrypoint* keyword in the list of declarations for that method.

Instantiation: Instantiation is linked to constructor methods. We can identify the instantiation of a static class (code that is executed when a static method of that class is called) and the instantiation of a normal class, i.e. the creation of an object. Both instantiation forms have a unique weave point at the point the first code inside the body of the constructor method is being executed. Because the call to the constructor for a normal instantiation is explicitly being made we can also identify two weave points in the method calling the constructor. Namely a weave point before the call to the constructor is made and a weave point after the call to the constructor is made. Note that in the first case the actual object to be created is not yet available for the weave point information. A normal constructor call is identified by a *newobj* instructions.

Method: For both static and non-static method calls, three weave points can be identified: before the call is made (in the calling method's body), just before the first code in the method body, and after the call is made (again in the calling method's body). Calls to methods are identified by *call* and *callvirt* instructions.

Field access: Field access can be split into setting the value of the field and retrieving the value of the field (both have a static and non-static form). Just as with methods we can identify a weave point before and after the actual setting or retrieving of the value of a field. Field access is recognised by the *ldfld*/*ldsfld* and the *stfld*/*stsfld* instructions.

Exception handling: Exception handling blocks are identified in the CIL code by the *.try* keyword. Following the *.try* keyword additional blocks can be identified by the *catch*, *filter*, *finally*, and *fault* keywords.

6.3 From crosscutting locations to weave points

Based on the list of crosscutting locations we have tried to make a mapping to the weave points we have identified. This mapping is listed in table 6.3. The list is split up in dymanic crosscutting locations and static crosscutting locations.

6.4 Supported weave points

In the first version of the weaver tool, not all weave points will be supported. The list of supported weave points is primarily based on the basic weave points needed to support an AOP implementation, like Compose*.
In that context the dymanic weave points that at least have to be supported are application start, object initialization, class initialization, constructor execution, method call, and field access. To reduce the complexity for the first version the matching will be done by name only, it will not

Crosscutting location	Concern implementation	Weave points	Remarks
Dynamic crosscutting (join points)			
Application start	after	entrypoint method	
Object initialization	before, after, around	object instantiation	constructor call, name or signature match
Class initialization	before, after, around	class instantiation	static constructor
Constructor execution	before, after, around	constructor body	name or signature match
Method call	before, after, around, replace	method invocation	name or signature match
Method execution	before, after (3 cases), around, replace	method body	name or signature match
Field access (get and set)	before, after, around, replace	load and store field	name or signature match
Exception handling	before, after, around, catch	throw, catch block	exception name/type used to identify
This	before, after, around	loading *this*	*this* object identified by its type
Target	before, after, around	loading object	*target* object identified by its type
Arguments	before, after, around, replace	loading sequence of vars	only applicable to constructors and methods
Control flow (if)	before, after, around	other join point	combined with other join point(s) to guard execution
Dataflow [30]	after	sequence of other join points	
Assignment	before, after, around, replace, catch	all store operations	
Static crosscutting (inter-type declarations)			
Add constructor	new constructor specification	class specification	limited to a new signature
Add static constructor	new constructor specification	class specification	only if a static constructor does not yet exist
Remove constructor	n/a	class specification	identified by signature (non-static)
Add method	new method specification	class specification	
Remove method	n/a	class specification	
Add field	new field specification	class specification	
Remove field	n/a	class specification	
Add attribute	new attribute specification	many points	
Remove attribute	n/a	many points	
Inheritance relationships	before, catch	object instantiation, casting	support for multiple inheritance

Table 6.1: Mapping of crosscutting locations to weave points.

be possible to match on signature.
Static weave points (based on inter-type declarations) will not be available in this first version of the weaver tool.

6.5 Summary

This capter described the mapping of AOP constructs, the crosscutting locations, to well-defined points in the CIL, so called weave points.
The crosscutting locations can be split up in *dynamic crosscutting locations* and *static crosscutting locations*. Dynamic crosscutting locations are well-defined points in the execution of a program, static crosscutting locations are well-defined points in the type structure of a program. In both types of crosscutting locations *primitive crosscutting locations* can be identified, this is a minimal set of crosscutting locations in which all crosscutting locations can be expressed.
The concern implementation can be added when a crosscutting location is identified. We identified four possible ways to add the concern implementation: insertion before the crosscutting location, insertion after the crosscutting location, insertion around the crosscutting location, and replacement of the crosscutting location.
The well-defined points in the CIL, the weave points, can be split up in static and executional weave points. The identified static weave points are: contructor, method, field, attribute, delegate, property, inheritance relationship. Executional weave points identified are the following: application start, instantiation, method, field access, exceptional handling.

CHAPTER 7

Towards a solution

This chapter will start with an overview of related work in section 7.1. Based on the findings we will describe four possible approaches to weave aspects into CIL code: source code weaving, CIL code weaving using the profiling APIs, CIL code weaving using assemblies, and adapting the Common Language Runtime (CLR). The first approach, described in section 7.2, is based on a standard representation for languages conforming to the Common Language Specification (CLS). The second approach relies on the modification of the CIL code just before it is compiled into native machine code. To modify the code at runtime we have to use special hooks into the CLR. This approach is described in more detail in section 7.3. The third approach involves the adaptation of the CLR itself, and will be described briefly in section 7.4. The fourth approach, which can be used to weave the aspects into the program at compile-time, is the modification of .NET assemblies. Section 7.5 will explain this approach in more detail. Finally section 7.6 will give a short summary of the different approaches explained in this chapter.

7.1 Related work

In this section first a brief overview will be given of currently existing AOP approaches targeting the .NET Framework. Some of these approaches are based on modifying the .NET intermediate language code. As some Java AOP approaches are also based on modifying the Java intermediate language code, this section will also look at some of the Java tools used for byte-code manipulation as reference.

7.1.1 AOP Solutions for the .NET Framework

Aspect#
Aspect# is a framework for the CLI, based on the usage of *DynamicProxies* [7]. This approach relies on the program developer to include references to the Aspect# framework. For example the creation of the Aspect# engine, or the implemention of Aspect# interfaces.

LOOM .NET

LOOM .NET is an aspect weaving tool, which weaves aspect code into already compiled .NET assemblies [50]. To inspect the already compiled .NET assemblies LOOM .NET uses meta data and reflection mechanisms. Based on the original classes, contained in the .NET assemblies, *proxy classes* are created during the aspect weaving process. A *proxy class* contains the aspect code and can be compiled to produce an extended version of the original class. These extended classes are linked together to reproduce the original structure contained in the .NET assembly. Because weaving is done on already compiled .NET assemblies LOOM .NET works implementation language independent.

Listing 7.1 shows a typical class template. The *classprotection* (the modifiers of the class), *classname* (the name of the class), and *baseclass* (the base class) symbols are used to identify the original class. For each identified class, a proxy class is created. The extensions to the class are defined in the *memberdefinition* symbol.

Listing 7.1: A LOOM .NET class template [50, pg. 3].

```
1 /*[CLASSPROTECTION]*/ class /*[CLASSNAME]*/:/*[BASECLASS]*/
2 {
3 /*[MEMBERDEFINITION]*/
4 }
```

Listing 7.2 shows an example of an interception using the method rule. The *modifier* (the modifier of the method), *resulttype* (the result type of the method), *methodname* (the method name), and *paramdeclaration* (the argument declaration of the method) symbols are used to identify the method(s). The method in the proxy class will call the *Log* method of the *MyLogger* class, before executing the original implementation.

Listing 7.2: A LOOM .NET method rule [50, pg. 4].

```
1 public /*[MODIFIER]*/ /*[RESULTTYPE]*/ /*[METHODNAME]*/(/*[PARAMDECLARATION]*/)
2 {
3   MyLogger.Log("enter /*[METHODNAME]*/");
4   /*[RETVALINIT]*/
5   /*[RETVALASSIGN]*/ base./*[METHODNAME]*/(/*[PARAMLIST]*/)
6   /*[RETVALRETURN]*/
7 }
```

Microsoft Phoenix

Phoenix is a framework designed to support compilation and program analysis techniques [40]. The framework provides building blocks, which can be linked together to cover the entire compilation process. This compilation process also includes analysis and optimization tasks. The architecture of Phoenix is shown in figure 7.1.

To handle different implementation languages, Phoenix uses an intermediate representation (Phoenix IR) to represent the CIL code. All analysis, instrumentation, and optimization tools target this IR, and in doing so are independent of the implementation language.

An AOP solution can be implemented as different building blocks targeting the Phoenix IR. Weaving the crosscutting locations and aspects can be done by modifying the Phoenix IR.

SourceWeave.NET

SourceWeave.NET is cross-language source code weaver [24], i.e. crosscutting concerns can be introduced independent of implementation language. The joinpoint and aspect model used are based on AspectJ. Pointcuts and advice are defined in a separate XML specification.

The SourceWeave.NET architecture, see figure 7.2, consists of three main components: a parser

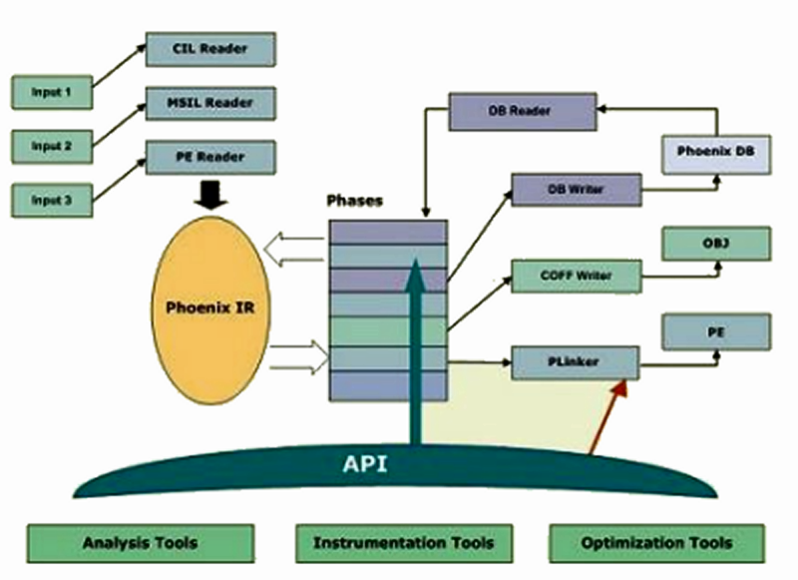


Figure 7.1: The Phoenix compiler platform.
Source: Phoenix Framework [40].

component, a joinpoint model component, and a compilation component. The *parser component* consists of a set of language parsers that convert the source code to the SourceWeave.NET AST. The AST used by SourceWeave.NET is based on the CodeDOM (Code Document Object Model). Note that each implementation language to be supported requires a specific language parser. The *joinpoint model component* weaves the source code and the aspects. Weaving is done on the AST created by the *parser component*. Finally, the modified AST is compiled into assemblies by the *compilation component*.

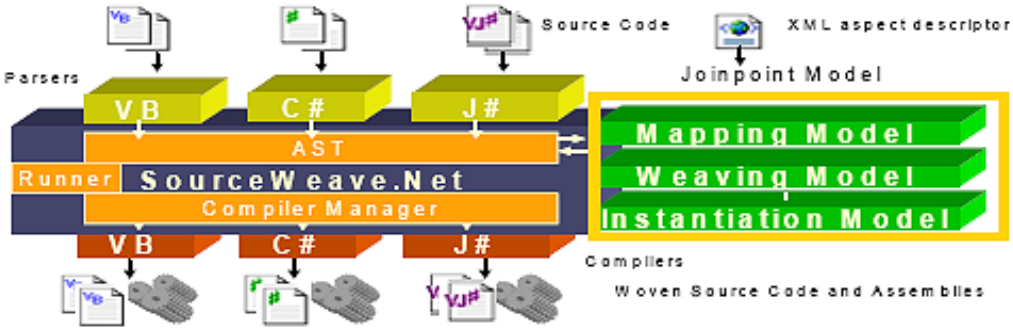


Figure 7.2: The SourceWeave.NET architecture.
Source: SourceWeave.NET: Cross-Language Aspect-Oriented Programming [24].

Benefits of SourceWeave.NET are:

1. The separation of the AOP XML language from the CLS. Allowing the usage of an extensive joinpoint model and providing more reusable code and crosscutting specifications.
2. Cross language-support by weaving on an AST. Any implementation language that has a

CodeDOM parser can be woven. Although the CodeDOM itself, it is a very generic AST, places some limitations on the cross language-support as described by the limitations below.

3. Native debugger support. Because the aspects are woven into the source code, the developer can trace through the execution of the woven code using the native debugger for the implementation language the original code was written in.

Limitations of SourceWeave.NET are:

1. A parser from the .NET language to CodeDOM is needed.
2. Limitations in expressiveness of CodeDOM. Not all .NET languages' constructs can be mapped to CodeDOM, for example nested namespaces.
3. Weaving can result in unsolvable compilation dependencies. In other words, the source code can not be compiled as a circular dependency is introduced after weaving. This problem is inherent in many AOP solutions.
4. Access to source code is required. Hence pre-compiled, third-party components cannot be included in the weaving process.

Weave.NET
The programming model used by Weave.NET [28] addresses two issues: how to specify aspects, and what architecture to use to compose the aspects with the components.
To specify aspects, Weave.NET uses the semantics from AspectJ, but AO cross-cutting details are defined in a separate *XML deployment script*. This allows the aspect behavior and components to be implemented in any language targeting the CLI.
The heart of the Weave.NET architecture is formed by the *Weave.NET Tool*, see figure 7.3. Provided a component and the crosscutting specifications, the Weave.NET Tool will modify the code of the component to bind join points to aspect behavior. Binding join points is done by introducing method calls to the aspect behavior, creating a *woven component*.

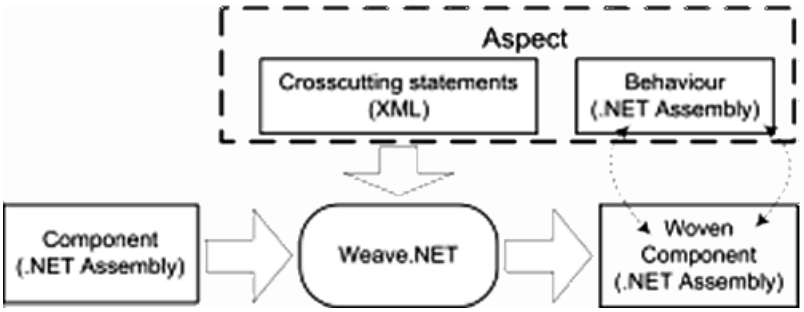


Figure 7.3: The Weave.NET architecture.
Source: Language-Independent Aspect-Oriented Programming [28].

7.1.2 Code-manipulation Tools

BCEL
The Byte Code Engineering Library (BCEL) allows analysis, creation, and manipulation of

binary Java class files [57]. The symbolic information of the class, e.g. methods, fields, and byte code instructions, is represented by objects, which can be manipulated. BCEL only provides a way to read, manipulate, and write Java class files, which manipulations to perform is up to the user.
A tool like BCEL can be used to weave AOP aspects into Java classes, for example AspectJ and JMangler [16] use BCEL as weaver. In .NET terms this would mean a tool to read, modify, and write the CIL code. The CIL code is read for example from an assembly. Namespaces, classes, methods, etc. would be represented by objects for easy manipulation. After manipulation, a new assembly with the updated CIL code has to be constructed.

Javassist
Like BCEL, Javassist [8] is also a toolkit for transforming Java class files. But, unlike other libraries, Javassist is a reflection-based toolkit. This means users do not have to have detailed knowledge of bytecode as transformations are described with source-level vocabulary.
Describing transformations in an implementation language under .NET is less desirable, because breaking free of a single implementation language is one of the big advantages .NET offers in this context.

7.2 Approach 1: Source code weaving

The .NET architecture has the CLS defined for interoperability between components written in different implementation languages (see chapter 3 for more information). Based on the CLS an AOP solution can define crosscutting locations. At these crosscutting locations, the AOP solution can use the code generation abilities of the .NET Framework to reflect AOP behavior in the source code. Afterwards, the modified source code can be compiled with the native compiler for the implementation language.
An example of an approach aimed at source code weaving under .NET is SourceWeave.NET [24].

7.2.1 Advantages and disadvantages

As with all solutions based on source code weaving, one of the biggest advantages is the use of the language specific compiler to compile the code after weaving. The major disadvantage of this approach is the need to create a custom source code parser to support a language. This source code parser is needed to create a generic AST, representing the source code, on which the weaving can be done.

7.3 Approach 2: Weaving at run-time with the profiling APIs

To monitor the performance and memory usage of programs executing on the Common Language Runtime (CLR), special interfaces can be used. These interfaces, the profiling APIs, provide an efficient way to hook into CLR events such as execution entering or leaving a method. These hooks into the execution of a program are intended as a way to measure the performance of parts of a program. But instead of adding code to measure the performance, we can also add other code. In this way we can use the hooks to execute aspect code. One of the simplest cases is hooking into the method entering and leaving events to provide before and after advice on

a method. More advanced cases could hook into the JIT compilation events, analyse the MSIL code stream, and insert additional code into the original code stream. For example this approach could be used to scan the code stream for calls to certain constructors or methods and execute something else first.
Before explaining the possibilities of these profiling APIs to weave aspects at run-time in more detail, we will first briefly describe the profiling APIs.

7.3.1 The profiling APIs explained

The profiling APIs are implemented as two COM interfaces, shown in figure 7.4. One is implemented by the CLR (ICorProfilerInfo), the other is implemented by the profiler (ICorProfilerCallback).
The *ICorProfilerCallback* interface is the 'notification API', consisting of methods with names

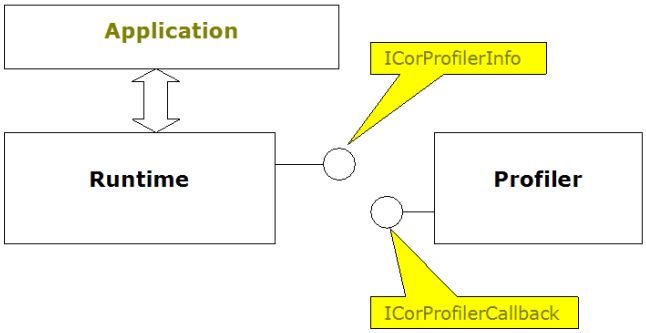


Figure 7.4: The two COM interfaces of the profiling APIs.
Source: Profiling [31].

like ClassLoadStarted, ClassLoadFinished, FunctionEnter, FunctionLeave. Each time the CLR takes one of these actions it calls the corresponding method in the profiler's *ICorProfilerCallback* interface.
The other profiling interface is *ICorProfilerInfo*. This interface exposes methods the profiler can use to get more information about the CLR event being analysed. For example, the CLR supplies the profiler with a *FunctionId* of the method being entered. With this *FunctionId* the profiler can call the *ICorProfilerInfo* interface to discover more information about the method identified with that *FunctionId*.
Loading a profiler is done by the CLR itself during its initialization in each process. Depending upon the value for two environment variables (Cor_Enable_Profiling and Cor_Profiler) the CLR decides whether or not to load a profiler. Note that this allows the usage of only one profiler for a process at one time in a given environment.

7.3.2 Implementing an aspect profiler

The implementation of the base framework for a profiler is straight-forward, as multiple code samples are available. But the power, and hardest goal to achieve, of the aspect profiler is the analysis and modification of the MSIL code stream before it is JIT compiled. Achieving dynamic run-time weaving should be possible with this approach. Code manipulation occurs in memory, leaving the original code on disk. When updating the aspect specifications, the

aspect profiler should mark methods for recompilation. Next time the method, is executed the aspect profiler can modify the code stream according to the new aspect specifications. Unfortunately, the analysis of the code stream is complex. There are two main reasons for this. The first reason is the mapping that has to be made between the way the CLR identifies constructs like classes, and methods and the way one could identify these constructs outside the CLR to create the aspect specifications. E.g. the CLR identifies a method with a unique *FunctionId* which only has a meaning inside the CLR at that moment while outside the CLR the same method can only be identified by another ID referring to the method's name. The second reason that complicates the process is the fact that operations are performed at the lowest possible form of MSIL outside of the .NET Framework. This means that tasks normally left to compiler tools have to be programmed yourself (e.g. if-statements do not exist, you really have to write out the correct conditional branch statements) and that really useful functionality like .NET Reflection is not available.
To illustrate one of the more complex problem occurring in the manipulation of the code stream we will look at jumping and branching statements. As mentioned in section 5.4.3, a jump or branch is given as a positive or negative offset from the current position in the code stream. If we insert additional instructions in the code stream we have to scan the entire code stream for jump or branch instructions and if nessesary compute new offsets. Worst case scenario also involves expanding short forms to normal forms accommodating a new offset which exceeds the size a short form can hold. If such an expansion occurs, the scan process has to be restarted, because this expansion can make the offset of a branch statement already scanned invalid. Maybe it's wiser to expand all short forms by default, eliminating the need to restart the scan process. However these short forms exist for a reason, to save memory. To find the best possible solution in both memory usage and processing time for this problem more investigation has to be done. So apart from the need for a more complex modification scheme, manipulation time will increase. In other words runtime performance of the program will suffer from this.

7.3.3 Advantages and disadvantages

The profiling APIs provide a way to hook into CLR events and in doing so allow us to execute aspect code. The following listing will give the most important advantages of this approach:

Independent from the unit of deployment: As described in section 5.1 the assembly is commonly used to deploy .NET programs. By hooking into the CLR we let the CLR take care off loading the CIL codestream from disk, for example read the contents of an assembly.

Dynamic weaving possible: Only the code in memory is modified to include the aspects. The CLR provides a hook to mark certain code for recompilation, this makes it possible to add or remove aspects from an executing program.

Of course the approach with the profiling APIs has a few major disadvantages too:

Running a profiler requires administration privileges: To register a profiler with the Windows operating system administration privileges are required.

Profiling interface allows only one active profiler: The profiler and the program are linked one-to-one. This means only one profiler can be active for a program. Using the profiling

approach to weave aspects into the program means no performance measuring tool using the profiling APIs can be used.

A profiler runs outside of the .NET CLR: A profiler hooking into CLR events runs outside of the .NET CLR itself. To be exact a profiler runs as a COM object in the process space of the operating system.

Negative impact on the runtime performance of a program: Since all the modification are done at runtime the program suffers without doubt performance loss.

7.4 Approach 3: Adapting the Common Language Runtime

The third approach involves adapting the runtime environment itself. The mechanism to enforce aspect behavior is built into the runtime. The aspect behavior and the crosscutting specifications have to be provided separate of the base program to the runtime.

7.4.1 Advantages and disadvantages

Major advantage of this approach is the seamless integration of the AOP solution with the runtime environment. Errors resulting from the AOP added behavior can be caught and handled by the runtime environment. The downside of the this integration with the runtime environment is the fact that to run a program the AOP modified runtime environment is needed. This also puts the additional task upon the AOP solution developers to keep their modified runtime environment up-to-date. Modifying an open-source runtime environment would be the easiest solution. Unfortunately, the only state-of-the-art CLR is the .NET Runtime and not all of the .NET source code is publicly available. Adapting open-source code CLR projects like Mono ([41]) or ROTOR ([48]) is less viable as these projects are still *under development*, and the modified runtime environment has to be updated frequently.

7.5 Approach 4: Weaving aspects into .NET assemblies

The fourth, and last, approach is based on the modification of .NET assemblies. At compile-time we can modify these .NET assemblies generated by the different language compilers and include code statements to execute our aspects. In principal this is the same idea as the aspect profiler approach. But at compile-time, the time needed to insert the statements is less of a problem (within certain boundaries of course). Another advantage compared to the aspect profiler approach is that we are now able to use .NET Reflection to find information or to create constructs for us. Unfortunately, .NET Reflection works at a higher level of abstraction than the profiling APIs do, resulting in the fact that we have no easy access to the MSIL code stream (section 7.5.1 will discuss this problem in detail). Since we are weaving assemblies and storing the modified assemblies to disk we have to consider a few problems that might arise from this, section 7.5.2 will explain those problems in more detail.

7.5.1 Getting the MSIL code out of the assembly

As mentioned before, .NET itself does not provide a way to easily access the MSIL code stream of an assembly. The .NET Reflection methods only provide an easy way to access the metadata of the assembly. But even with this metadata info it is not possible to extract the code stream. As explained in section 5.1.1, all data is stored in a PE file, and to extract the code stream you have to get the correct offsets and lengths for a particular code stream. This leaves us with the following options:

Build a PE file reader/writer Requires a thorough understanding of the PE file format and will result in a lot of work before we can get to the real challenge, the weaving of the MSIL code. Another disadvantage of this approach is that changes to the PE file format for a next .NET Framework version have to be integrated.

Use a third-party PE file reader/writer If we can use an open-source PE file reader/writer for .NET, we may be able to quite easily extend it with a MSIL code stream manipulation method. Unfortunately at the time of investigation no such tool existed. Only a few demonstrating examples were available and in practice we would be building our own PE file reader/writer tool.

Using the ilasm and ildasm tools These .NET supplied tools are a fully functional IL Assembler (ilasm) and an IL Disassembler (ildasm). Using *ildasm* it is possible to disassemble an assembly, creating a textual IL representation of the .NET code contained in the assembly. With *ilasm* we can assemble a textual IL file and create an assembly. Now we only have to build a tool that can read textual IL, apply the manipulations and write out the modified textual IL.

A clear distinction between the first two approaches and the third approach is the manipulation of respectively binary IL and textual IL.

7.5.2 Problems with weaving assemblies

The following issues have to be considered in relation to weaving assemblies:

Manipulating signed assemblies: To protect the assembly against modification and to verify that the assembly is authentic an assembly can be *signed*. Signing an assembly is done by a private key mechanism. The private key, only known to the signer of the assembly, is used to generate a signature, *the public keytoken*, for the assembly. All assemblies that reference a signed assembly will include the public keytoken of the signed assembly. The security system of .NET will check if the public keytoken of a referenced assembly is the same as the expected keytoken. In other words we can never modify an assembly to include additional code for aspects, because we do not have the private key to sign the assembly.

Manipulating the standard .NET libraries may not be feasible: A few practical problems arise when manipulating standard .NET libraries. The huge size of some of these libraries could drastically impact performance. But this also means that your application has to be distributed with its own version of some standard libraries. Which means updates to these standard libraries are not automatically integrated in your application, we

have to recompile and redistribute for that. Apart from the practical issues involved one could argue about the desirability of modifying standard libraries; what is the meaning of 'standard' in this context?

Manipulating developer supplied assemblies: These are the third-party assemblies the developer includes in his project. Just like the standard .NET libraries, we have to distribute our manipulated assemblies along with the application. If there is an update of a third-party assembly we have to recompile and redistribute our application to incorporate the changes. Furthermore we have to consider the legal issues when modifying third-party assemblies. The developer may very well have bought the assembly and is allowed to use it, but not to change it. So is it legal if we modify such assemblies and they are distributed by the developer in a commercial application?

7.5.3 Advantages and disadvantages

As every language targeting .NET Framework is compiled into the CIL, this approach abstracts from any source code knowledge. Apart from multi-language support, being able to weave third-party assemblies is a clear advantage. Although weaving third-party assemblies is not without any problems, as described in section 7.5.2. Disadvantage of weaving assemblies is the loss of the link with the source code for a debugger.

7.6 Summary

In this chapter various approaches for implementing an AOP solution in .NET have been discussed. The major solutions discussed in section 7.1.1 are *LOOM .NET*, *SourceWeave.NET* and *Weave.NET*. As modifying the Java byte-code to support aspects is used in many AOP solutions for the Java platform, section 7.1.2 briefly describes a few byte-code manipulation tools for the Java platform.
Sections 7.2, 7.3, 7.5, and 7.4 describe the four possible ways to implement an AOP solution in the .NET Framework: source code weaving, run-time weaving using the profiling APIs, adapting the .NET CLR, and weaving assemblies. For interoperability components, written in different implementation languages, have to conform to the CLS. An AST based on the set of language constructs defined in the CLS can be used in a cross-language source code weaver. Weaving at run-time can be done by using the special *profiling hooks* provided by the CLR for performance and memory usage measurement. It is also possible to integrate the AOP behavior into the runtime itself, this involves creating or adapting a runtime environment. Every application written in a .NET language is compiled to an intermediate language, the CIL. This CIL can serve as the input for a implementation language independent AOP weaver.

CHAPTER 8

The implementation of the CIL Weaving Tool

A detailed description of the implementation of the weaver tool will be presented in this chapter. Section 8.1 begins with an overview of the different parts of the weaver tool. Also the data flow between the environment and the weaver tool, and the data flow between the different parts of the weaver tool are given. Section 8.2 will discuss the input supplied by the weave specifications file. Sections 8.3 and 8.4 explain the two major parts of the weaver tool: the PE Weaver respectively the IL Weaver A short summary of the implementation of the weaver tool is given at the end of this chapter, in section 8.5.

8.1 Global structure of the weaver tool

The weaver tool consists of two major components, the PE Weaver and the IL Weaver. The PE Weaver (see section 8.3 for a detailed description) performs two steps. In the first step it converts an assembly into a textual IL file, this process is called disassembling. To process these texutal IL files, the IL Weaver is called. In the last step the PE Weaver converts the modified textual IL file to an assembly, this process is called assembling. Figure 8.1 shows this process, including the input and output of the data.

The separation of the assembling/disassembling process and the actual weaving process into two separate tools is the result of a limitation in the .NET Framework. Figure 8.1 shows that the *AssemblyInspector*, part of the IL Weaver, depends on the input assemblies. These assemblies are needed to gather information for the weaving process and are loaded into the application domain using .NET reflection. Unfortunately the .NET Framework has no method to unload an assembly from the application domain. This results in assemblies being locked by the operating system as long as the application runs. But one of the loaded assemblies may very well be an assembly we have to modify, which is not possible since it is locked (i.e. we cannot overwrite the file with the modified version). In the solution with two applications the assemblies are only loaded into the IL Weaver application domain and hence they are unlocked when the IL Weaver exits. Only after the IL Weaver has exited the PE Weaver will start overwriting the old

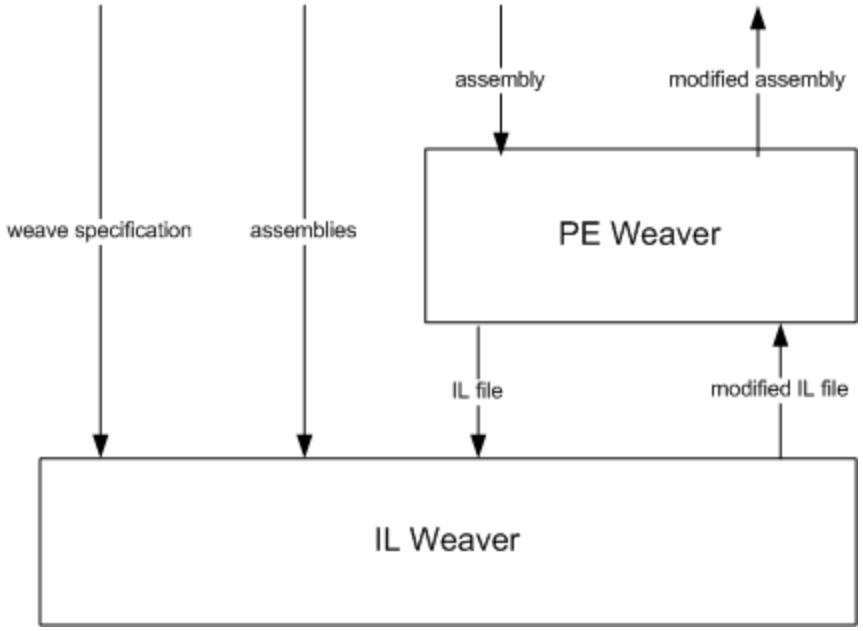


Figure 8.1: Data flow diagram weaver.

assemblies with the new modified versions.

8.2 The weave specification file

To specify a set of weave operations, a separate configuration file is used, see appendix C for a complete listing of the layout of this file. This weave specification file is an xml file. Using a xml format has two advantages. First, the content of the file is relatively easy to understand for the developer incorporating the weaver in his own project. After all, he has to make a mapping from the weave operations native to his project to the format used by the weaver. Secondly, using a file layout as xml allows for easy extensions of the weaver. The extension of the weaver will require additions to the weave specification file to specify this new or updated functionality.

In listing 8.1 the general structure of the weave specification file is given. The weave specification has been given a version number to identify different versions. The first block in the specification file lists references to assemblies that should be included or excluded in the to be woven assembly, section 8.2.1 will explain this block. The second block contains all the method definitions, which can later be referenced in the application and class blocks. A more detailed description of the method definition block can be found in section 8.2.2. The third block specifies modifications at the application level, for example executing a method at the start of the application. Section 8.2.3 will describe the application block in more detail. The fourth, and last block specifies all the modifications at the class level. A class block is identified by the fully qualified name of the class it should be imposed on. This means that the class block is not unique, multiple class blocks with different values for the fully qualified name can occur. The details of the class block are discussed in section 8.2.4.

Listing 8.1: The weave specification file

```
<?xml version="1.0" encoding="UTF-8"?>
<weaveSpecification version="<weave specification version>">
  ["assembly reference block"]
  ["method definition block"]
  ["application block"]
  ["class block"]
</weaveSpecification>
```

8.2.1 The assembly reference block

Each assembly can contain references to other assemblies. The CLR uses these references to check if everything is available to run. When adding code, for example method calls, it can occur that the target is defined in an external assembly which was not yet referenced. For such cases the assembly reference block has the *forceReferenceIn* attribute. To force a reference in all assemblies, the wildcard cam be used. It can also occur that all code that uses a certain assembly is removed, making the assembly reference obsolete. The attribute *remove* provides a way to remove certain assembly references, so unused assemblies no longer need to pass the CLR checks. Listing 8.2 shows the structure of the assembly reference block.

Listing 8.2: The assembly reference block

```
<assemblies>
  <assembly name="<assembly name>"
            version="<assembly version>"
            publicKeytoken="<keytoken of the assembly>"
            forceReferenceIn="[<assembly name>/*]"
            remove="[yes/no]"/>
</assemblies>
```

Example: If we introduce a logging aspect to an existing program, we have to make sure the .NET Runtime can find our logging methods. These logging methods are supplied in an extra assembly, called *LoggingLibrary*. This assembly is identified by its name (name attribute) and by its version number 1.0.0.0 (version attribute). But the program in which we introduce the logging, has no knowledge of this assembly. Using the assembly reference block we can create references in all the assemblies of the program to this *LoggingLibrary* by using the wildcard for the forcereRerenceIn attribute. See listing 8.3 for the example code.

Listing 8.3: Referencing the *LoggingLibrary*

```
<assemblies>
  <assembly name="LoggingLibrary"
            version="1.0.0.0"
            publicKeytoken=""
            forceReferenceIn="*">
</assemblies>
```

8.2.2 The method definition block

The method definition block defines signatures of methods, which can be used in the application and class blocks. Signatures have to reflect methods that exist in the application domain. A overview of the method definition block can be found in listing 8.4.

Listing 8.4: The method definition block

```
<methods>
```

```
  <method id="<id of the method reference>"
          assembly="<assembly the method can be found in>"
          class="<fully qualified name of the class the method belongs to>"
          name="<name of the method>"
          returnType="<type of the function result>">
    <argument value="" type="[string/int]"/>
    <argument value="%senderobject"/>
    <argument value="%createdobject"/>
    <argument value="%targetobject"/>
    <argument value="%targetmethod"/>
    <argument value="%originalparameters"/>
    <argument value="%casttarget"/>
    <argument value="%fieldvalue"/>
  </method>
</methods>
```

Within the method definition block more than one method definition can be defined. Each method definition consists of four attributes to identify the method and an optional number of attributes defining the parameters of the method.
The first attribute is the *id*, this is the name this method can be referenced by in this weave specification file. The second attribute, *assembly*, contains the name of the assembly the linked method belongs to. The third attribute, *class*, defines the fully qualified name of the class this linked method is part of. The fourth attribute, *name*, is the name of the method that is being linked. Together the *assembly*, *class*, and *name* attributes uniquely identify a method.
To define the parameters of the linked method an optional number of *argument* attributes can be supplied to the method definition. There are two options to define a value for a parameter: a constant value, or a dynamic value based on the runtime context where the method is called. Currently the only constant values supported are the string and integer types. The currently supported dynamic values are explained in more detail below:

%senderobject (type *object*): The object in which the weave point is found is passed to the target function. Note that this has only any meaning in a non-static context, i.e. the weave point is found in a non-static context. When used in a static context, a *null* value will be passed to the target function.

%createdobject (type *object*): This value can only be used in the context of an *after class instantiation block* (see below for a description). At runtime the newly created object instance will be passed as parameter of the target function. The parameter type for this parameter of the target function is the *object* type. There are two reasons to use the most basic type, *object*, for this. First, the target function can now be used in more than one class block. This allows the developer to create a single function for handling the class instantiation, for example logging the creation of new objects with the type of the object as extra information. Second, it allows the developer to separate the application from the methods handling the weave points. Which in turn allows easy re-use of weave point handling methods by putting them in a seperate assembly, independent of the application.

%targetobject (type *object*): If the weave point is a non-static operation on an object, e.g. a method call, the object can be passed to the target function. If no object is available at the weave point, a *null* value will be passed to the target function.

%targetmethod (type *string*): If the weave point is a method call the name of the method can be passed to the target function using the *%targetmethod* value. This can be used when the weave point is very general, for example all calls to methods of a certain object. The

target function can now be supplied with the exact name of the called method intercepted at that point.

%originalparameters (type *object array*): In case the weave point is a method call, the *%originalparameters* value can be used to pass all the parameters of the original method call to the target function. The original parameters are placed in an array and are passed as a single parameter to the target function.

%casttarget (type *string*): If the weave point is the cast from one object type to another, the target function can be supplied with the object being casted and the type casted to. The object being casted can be passed to the target function using the *%targetobject* value. To pass the type casted to the *%casttarget* value can be used. The type of this value is the *string* type, in other words the target function receives the name of the type being casted to.

%fieldvalue (any type): If the weave point is an operation on a field, the original value of the field can be passed to the target function with the *%fieldvalue* value.

Example: The *LoggingLibrary* assembly contains the class *FileLogger* to log events to a file. One of the methods of the *FileLogger* class is the *Log* method, the signature of the *Log* method can be found in listing 8.5. The *Log* method has four parameters: the sender object (the object that made the intercepted call), the target object (the object on which the intercepted call was made), the name of the method that was intercepted, and the parameters passed to the method that was intercepted.

Listing 8.5: The *Log* method of the *FileLogger* class

```
1  public class FileLogger
2  {
3    public void Log(Object sender,
4                    Object target,
5                    String targetmethod,
6                    Object[] parameters) {
7    }
8  }
```

Listing 8.6 shows the reference, in the weave specification file, to this *Log* method. In the rest of the weave specification file this *Log* method can now be referenced with the id *logOpenConnection*. To pass the required values to the *Log* method the corresponding four dynamic values are used in the method definition: *%senderobject*, *%targetobject*, *%targetmethod*, and *%originalparameters*.

Listing 8.6: The method definition of the *Log* method

```
1  <methods>
2    <method id="logOpenConnection"
3            assembly="LoggingLibrary"
4            class="FileLogger"
5            name="Log"
6      <argument value="%senderobject"/>
7      <argument value="%targetobject"/>
8      <argument value="%targetmethod"/>
9      <argument value="%originalparameters"/>
10   </method>
11 </methods>
```

8.2.3 The application block

The structure of the application block is given in listing 8.7. Below is an explanation of the different attributes.

As identified in chapter 6 (see the dynamic crosscutting locations in table 6.3) the start of the application is a weave point. The application block allows to link a method, defined in the *method definition block*, to the start of the application using the *notifyStart* attribute.

Listing 8.7: The application block

```
<application name="">
  <notifyStart id="<reference to a method definition>"/>
</application>
```

Example: This weave point can be used to initialize the AOP environment, e.g. an interpreter, in which the program should be executed.

8.2.4 The class block

In listing 8.8 the layout of the class block is given. A weave specification file can contain multiple class blocks, each identified by a different class name in the *name* attribute. Apart from identifying a class block by the *fully qualified class name*, a wild card can be used as *class name*. A wild card matches every class in the application. For example this can be used to add a call to a logging method in every class at a file operation.

Listing 8.8: The class block

```
<class name="[*/<fully qualified class name>]">
  ["after class instantiation block"]
  ["method invocation block"]
  ["cast block"]
  ["class replacement block"]
  ["field access block"]
</class>
```

The class block contains four sub blocks, which will be explained below.

8.2.4.1 The after class instantiation sub block

The *after class instantiation block* contains a method reference in the *executeMethod* part. The id of the reference method is stated in the *id* attribute and refers to a previously defined method in the *method definition block* (see section 8.2.2). Listing 8.9 shows the *after class instantiation block*. This weave point is implemented in the method that instantiates the class, i.e. after a new object of the class has been created. Note that this allows us to pass the newly created object as parameter (using the %createdobject value) to the referenced method.

Listing 8.9: The after class instantiation block

```
<afterClassInstantiation>
  <executeMethod id="<reference to a method definition>"/>
</afterClassInstantiation>
```

Example: If we want to be notified of all *TcpConnection* objects being created by a program, we can add an after class instantiation block to the *TcpConnection* class. Listing 8.10 shows the related weave specification file parts.

Listing 8.10: Adding notification to *TcpConnection* object creation

```
<assemblies>
  <assembly name="LoggingLibrary"
            version="1.0.0.0"
            publicKeytoken=""
            forceReferenceIn="*">
</assemblies>

<methods>
  <method id="logNewConnectionObject"
          assembly="LoggingLibrary"
          class="FileLogger"
          name="LogNewConnectionObject"
    <argument value="%senderobject"/>
    <argument value="%targetobject"/>
  </method>
</methods>

<class name="TcpConnection">
  <afterClassInstantiation>
    <executeMethod id="logNewConnectionObject"/>
  </afterClassInstantiation>
</class>
```

8.2.4.2 The method invocation sub block

As identified in chapter 6 the call to a method is a weave point. This weave point can be specified in the *method invocation block*, see listing 8.11. A *methodInvocations* node can contain multiple *callToMethod* sub nodes, each identified by a different fully qualified method name (the *class* and *name* attributes). Contained in the *callToMethod* node are the *voidRedirectTo* and *returnvalueRedirectTo* nodes. These nodes are used to specify the target method to call, identified by its *id* attribute.

Listing 8.11: The method invocation block

```
<methodInvocations>
  <callToMethod class="" name="">
    <voidRedirectTo id="<reference to a method definition>"/>
    <returnvalueRedirectTo id="<reference to a method definition>"/>
  </callToMethod>
</methodInvocations>
```

The reason for two nodes to identify the target method is due to the fact that a .NET method can have a return value. If the intercepted method does not have a return value, in other words has the return type *void*, the target method reference in the *voidRedirectTo* node will be used. Otherwise if the intercepted method has a return value the target method reference in the *returnvalueRedirectTo* will be used. As return type of the target method the *object* type can be used. The weaver will automatically cast it to the expected return type. Note that it is the responsibility of the target method to return a correct castable type.

Example: A method invocation can be used to add logging to certain method calls. The method call we want to log can be defined in the attributes of the *callToMethod* node. See listing 8.12 for an example of adding logging to the *Open* method of the *TcpConnection* class. As the logging library (*LoggingLibrary* assembly) is not referenced in the assemblies using the *TcpConnection* class an explicit reference to the *LoggingLibrary* is made. This reference can be found in the *assemblies* block. The method responsible for logging the information to disk is the *Log* method of the *FileLogger* class provided by the *LoggingLibrary* assembly. The *methods* block shows the

details on the reference definition of the *Log* method. See section 8.2.2 for an explanation of the argument values used for the *Log* method.

Listing 8.12: Adding logging to the *TcpConnection* class

```
<assemblies>
  <assembly name="LoggingLibrary"
            version="1.0.0.0"
            publicKeytoken=""
            forceReferenceIn="*">
</assemblies>

<methods>
  <method id="logOpenConnection"
          assembly="LoggingLibrary"
          class="FileLogger"
          name="Log"
    <argument value="\%senderobject"/>
    <argument value="\%targetobject"/>
    <argument value="\%targetmethod"/>
    <argument value="\%originalparameters"/>
  </method>
</methods>

<methodInvocations>
  <callToMethod class="TcpConnection" name="Open">
    <voidRedirectTo id="logOpenConnection"/>
  </callToMethod>
</methodInvocations>
```

8.2.4.3 The cast sub block

In listing 8.13 the layout of the *cast block* shown. The weave point identified by the *cast block* is the cast from one class to another class as supported by the class hierarchy of the .NET Framework. Matching will occur on the fully qualified name of the target class of the cast, defined with the *assembly* and *class* attributes of the *castTo* node. The only supported action at this weave point is the execution of a target method. A reference to this method is stated in the *id* attribute of the *executeMethodBefore* node. The signature of the referenced method is defined in the *method definition block*.

Listing 8.13: The cast block

```
<casts>
  <castTo assembly="" class="">
    <executeMethodBefore id="<reference to a method definition>"/>
  </castTo>
</casts>
```

Passing the original object, i.e. the object being cast, to the target function can be done by defining the *%targetobject* value for the target function. The target method is expected to have a *object* as return type, logically it should be castable to the expected target type. By defining the *%casttarget* value for the target function it is possible to supply this target function with the type being casted to. (see section 8.2.2).

Example: This approach allows the target function to modify the original object, which can be exploited to introduce multi-inheritance into the single-inheritance class hierarchy of the .NET Framework. In the .NET code the type checking on casting can be avoided by explicitly casting to the *object* type before the desired casting is done, e.g. *TypeA a = (TypeA)(Object)(TypeB)*. At runtime this code will give a casting exception, as *TypeB* is not castable to *TypeA*. But the

target function can now substitute the object of *TypeB* with a valid object of *TypeA* to allow multi-inheritance. Note that the entire mechanism to keep track of a multi-inheritance tree and provide the right object at every weave point is not provided by this weaving tool.
E.g. a platypus has characteristics of both a mammal and a bird. So we want the class *Platypus* to inherit from the classes *Mammal* and *Bird*. To implement this we define the inheritance from the *Mammal* class in the .NET class hierarchy (*class Platypus : Mammal*). The inheritance from the class *Bird* will we provided by the AOP implementation, by defining a cast definition for the cast to the *Bird* object. To use the inheritance relation from the class *Bird* the following code can be used: *Bird* bird = (Bird)(Object)platypus. The weaving tool will intercept this cast and call the defined target funtion for this cast, which in turn should filter out the multi-inheritance casts, and return a *Bird* object representing the *Platypus* object.

8.2.4.4 The class replacement sub block

The *class replacement block* is in fact a simple renaming operation. See listing 8.14 for the structure of the *class replacement block*. Bound to the parent class block operations performed on a certain class can be redirected to another class. Inside the *class replacement block* multiple replacements can be defined, i.e. it can contain more than one *classReplacement* node. The class to be replaced (or renamed) is identified by the *assembly* and *class* attributes of the *classReplacement* node. The name of the target assembly and class are defined respectively in the *assembly* and the *class* attributes of the *replaceWith* node inside the *classReplacement* node.

Example: Logging operations in an existing application can be redirected to a new logging class as long as the public interface of the new logging class matches the interface in the existing application.

Listing 8.14: The class replacement block

```
<classReplacements>
  <classReplacement assembly="" class="">
    <replaceWith assembly="" class=""/>
  </classReplacement>
</classReplacements>
```

8.2.4.5 The field block

Three different operation can be identified at the field access weave point. Field access is defined as a reading or writing the value of a field. The three operations are: calling a target funtion before the field is accesed. calling the target function after the field is accesed, and replacing the field access operation with a call to the target function. An overview of the field access block can be found in listing 8.15.

Listing 8.15: The field access block

```
<fieldAccesses>
  <field class="" name="">
    <callBefore id="<reference to a method definition>"/>
    <callAfter id="<reference to a method definition>"/>
    <replaceWith id="<reference to a method definition>"/>
  </field>
</fieldAccesses>
```

The *class* and *name* attributes of the *class* node are used to identify the fully qualified name of the field. Note that the *fieldAccesses* node can contain multiple *field* nodes for different fields. The method referenced by a *replaceWith* node has to have its *returnType* attribute defined. It is the responsibility of this target function to return a valid type, i.e. the type of the result should not cause a .NET type conflict with the original type of the field.

Example: The *TcpConnection* class has a field, *ConnectionString*, to store the connection string, i.e. the name of the target machine and the associated port number. To log the value of this connection string every time it is used we can use the weave specification listed in listing 8.16.

Listing 8.16: Logging the connection string

```
<assemblies>
  <assembly name="LoggingLibrary"
            version="1.0.0.0"
            publicKeytoken=""
            forceReferenceIn="*">
</assemblies>

<methods>
  <method id="logFieldAccess"
          assembly="LoggingLibrary"
          class="FileLogger"
          name="LogFieldAccess"
    <argument value="\%senderobject"/>
    <argument value="\%fieldvalue"/>
  </method>
</methods>

<fieldAccesses>
  <field class="TcpConnection" name="ConnectionString">
    <callBefore id="logFieldAccess"/>
  </field>
</fieldAccesses>
```

8.3 The PE Weaver

As shown in figure 8.1 the main tasks of the PE Weaver are disassembling the original assemblies and assembling the modified assemblies. But apart from these two main tasks the PE Weaver can do verification of the original and modified assemblies. A more detailed view of the PE Weaver can be found in figure 8.2.
A class diagram of the PE Weaver can be found in appendix D.1. The PE Weaver consists of two classes: *PeWeaver*, and *ProcessManager*. The *ProcessManager* class is used by the *PeWeaver* class to execute the external application to verify, disassemble, and assemble .NET assemblies.

8.3.1 Verification of the assemblies

To verify the assemblies before and after the weave process the *peverify tool* distributed with the .NET SDK is used. This tool checks the assembly for various errors, e.g. stack overflows or unknown method/assembly references. Especially in the development process of a tool using the weaver error checking can be very useful.

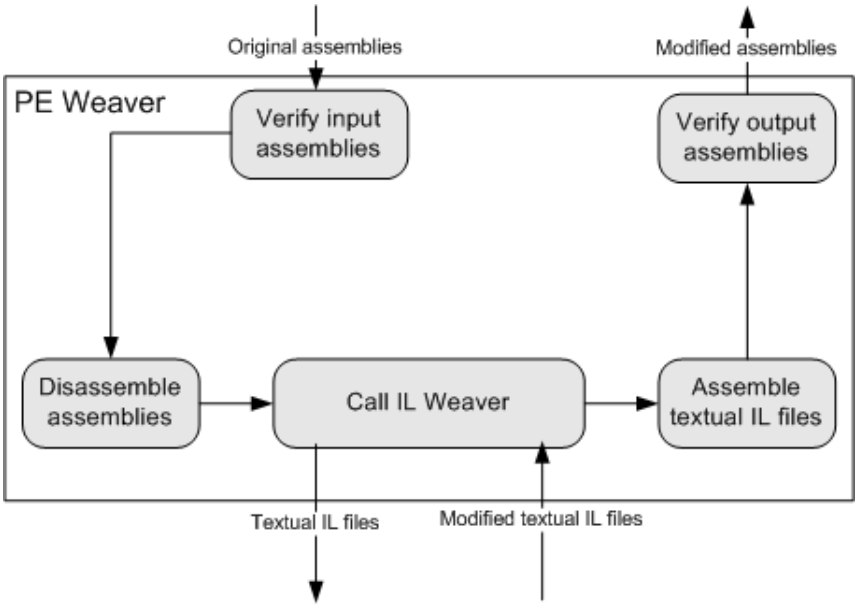


Figure 8.2: Data flow diagram PE Weaver.

8.3.2 Disassembling

Disassembling an original assembly is one of the main tasks of the PE Weaver. To disassemble an assembly the *ildasm tool* (IL Disassembler) distributed with the .NET SDK is used. The output of the *ildasm tool* is a text file with the textual representation of the IL code in the assembly. This textual IL file is used as input for the IL Weaver, see section 8.4.

8.3.3 Assembling

After the IL Weaver has modified the textual IL files a new assembly has to be created. This process is called *assembling* and is done by the *ilasm tool* (IL Assembler) part of the standard .NET Distribution. If the assembling succeeded without errors the original assembly will be overwritten, unless the special */out* command-line switch of the PE Weaver is used. With this switch it is possible to define the name of the new assembly and the original assembly will not be overwritten. This option is only usable when weaving a single assembly.

8.4 The IL Weaver

The IL Weaver performs the actual weaving of the instructions from the weave specification file into the IL code. Figure 8.3 shows the main tasks performed by the IL Weaver.

The input for the IL Weaver is the weave specification file, the textual IL file(s) containing the code (provided by the PE Weaver), and all assemblies needed to extract additional information about the .NET structure of the application. The output from the IL Weaver is a modified version of the textual IL file(s).
Additional information about these steps performed by the IL Weaver can be found below.

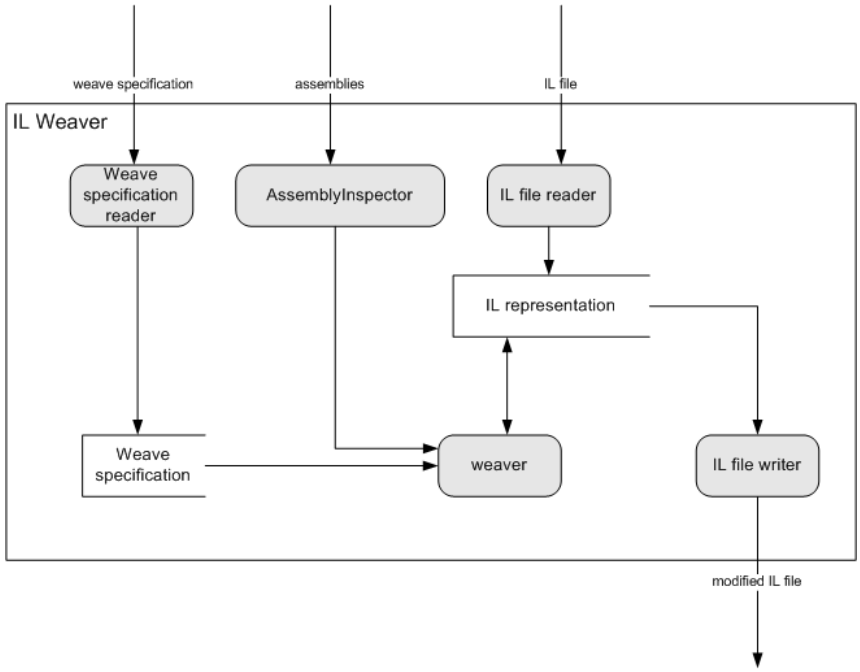


Figure 8.3: Data flow diagram IL Weaver.

8.4.1 Reading the weave specification file

In section 8.2 the detail of the weave specification file have been explained. To use the information stored in the weave specification file, the IL Weaver has to read it and store the information into an internal structure. This internal structure is needed to query the information fast.
An overview of the classes related to this internal representation of the weave specification file can be found in figure D.5 (appendix D).

8.4.2 Reading the IL file

Before the actual weaving can be done, the textual IL file generated by the PE Weaver has to be loaded. To be able to reason about the structure of the IL code contained in the textual IL file, the file is parsed into a special structure. This structure could in fact be a complete representation of the CIL language, but we haven chosen to only implement the part of the structure we need. We have chosen for this approach for the following reasons: simplicity, less error prone and upgradability. The first reason, simplicity, is because the CIL specifications is very detailed and complex. Besides implementing a structure to store every possible CIL option is not necessary, we only need a very limited part for the weave process. The second reason, less error prone, also results from the complexity of the CIL specifications. If we limit the structure to what we need and do not try to parse the rest, we only have to worry about a very small part of the complete CIL specifications. The third and last reason, upgradability, means that changes to the CIL specifications are less likely to have impact on the structure used to store the IL code.
The IL structure is located in the *Weavers.IlStructures* namespace and contains the following classes: NamespaceBlock, ExternalAssemblyBlock, ClassBlock, MethodBlock, and IlOpcode. A complete class diagram of the IL structure can be found in figure D.4 (appendix D).

8.4.3 The assembly inspector

The assembly inspector has the task to gather information about types in the application domain, when needed to perform certain weave operations. For example, checking inheritance relationships. The best way to get this information is using .NET Reflection. To be able to query information about a certain assembly using reflection the assembly has to be loaded into the application domain of the IL Weaver. To obtain the best performance, the assembly inspector uses an internal hashtable to store results, at the cost of increased memory usage. To keep the results available for the weave process of every IL file, the assembly inspector is implemented as a singleton.
Listing 8.17 shows the public interface of the assembly inspector. This *isMethod* method will iterate over the list of assemblies supplied by the *enumAssemblies* parameter and look for the class or type stated in the *className* parameter. If this type is found, it is retrieved with reflection using the *GetType* method. The result is stored in a *System.Type* object. The final step is invoking the *GetMember* method of the *System.Type* object to determine if the supplied *methodName* parameter is an existing method of this type.

Listing 8.17: Public method *IsMethod* of the assembly inspector

```
1 public bool IsMethod(String currentAssembly,
2                       IEnumerator enumAssemblies,
3                       String className,
4                       String methodName)
```

Especially in the process of determining the class hierarchy for casting operations this functionality is used. Because the weave specifications are defining a certain class to weave on, but in the case of methods inherited from a parent class, the IL code states the parent class. Matching just the class from the weave specifications to the class name in the IL code will not give the proper result.
For example: *ClassA* has a method *MethodOfA*, and *ClassB* inherits from *ClassA*. We want to intercept all calls made to an instance of *ClassB*, including the inherited method *MethodOfA*. But, the call in the IL code made on the instance of *ClassB* is *ClassA:MethodOfA()*. So, when a method call is found, in this case the *ClassA:MethodOfA()* call, a check has to be done to see if it can be a method call on an instance of *ClassB*. Otherwise the call to *MethodOfA* will not be found, as the *ClassB:MethodOfA()* call does not exist in the IL code.

8.4.4 Weaving

In the weave process an iteration over all the IL structures (see appendix D, figure D.4), based on the textual IL file, is performed. Inside the *MethodBlock* class, the IL instructions are stored in an array of *IlOpcode* objects. To identify weave points in the IL code the *IlOpcode* objects are checked for certain values, for example the opcode for a method call. When a weave point is identified, a weave operation can be performed, for example calling the defined target function. After all the weave points in a *MethodBlock* have been found, the weave process will update the IL administration of the method. In a number of cases the addition of calls to a target function require the increase of the maximum stack size value of the method.

8.5 Summary

In this chapter the implementation of the CIL Weaver has been discussed. The CIL Weaver is a combination of two tools: the PE Weaver, and the IL Weaver. The PE Weaver disassembles the input assemblies into textual IL files. These textual IL files are used as input for the IL Weaver. After the IL weaver finishes the weaving, the PE Weaver assembles the textual IL files into new assemblies. The IL Weaver performs the actual weaving and uses the following input: the weave specification, the textual IL files, and related assemblies to gather additional information. As output the IL Weaver gives a modified textual IL file, which can be assembled by the PE Weaver.
As stated earlier one of the goals of the CIL Weaver is to provide a tool that other developers can use to implement their own AOP tool. In the next chapter the integration of the CIL Weaver into the Composestar project is described. In a wider context this can be seen as an example of integrating the CIL Weaver into an AOP solution.

CHAPTER 9

Integrating the CIL Weaving Tool into Compose*

This chapter will show the integration of the CIL Weaving Tool into the Compose* project.
The integration of the CIL Weaving Tool is done in two steps. The first step is the creation of the weave specification file, which is described in section 9.1. The second step is running the CIL Weaving Tool and have it apply the needed modifications to the assemblies of the target application. This second step is described in section 9.2.

9.1 Creating the weave specification file

As described in chapter 8 the information on how to weave is supplied to the weaver tool by a file, the weave specification file. To integrate the weaver tool into an AOP project a component has to be made that generates this weave specification file.
Compose* itself has a modular design and the generation of the weave specification file can be added as a module. The name of this module is CONE (COde GEneration), more information on the structure of Compose* can be found in chapter 2.
The CONE module consists of two interfaces: *Composestar.Core.CONE.RepositorySerializer*, and *Composestar.Core.CONE.WeaveFileGenerator*. The first interface is for the generation of the repository file, and the second interface is for the generation of the weave specification file.
As the weaver tool described in chapter 8 is only usable for the .NET environment, the implementation resides in the *Composestar.DotNET.CONE* package. The class diagram of the *Composestar.DotNET.CONE.DotNETWeaveFileGenerator* class can be found in figure 9.1.

The Compose* master module invokes the *run* method, which calls the methods to write out the required weave specification file blocks. The blocks used are the assembly reference block, the method definition block, the application block, and the class block.

The assembly references:
First of all, the Compose* runtime assemblies have to be referenced, otherwise the calls to the runtime interpreter defined at the weave points will result in a runtime error. Secondly, in the main assembly of the application, all the assemblies making up the application have to

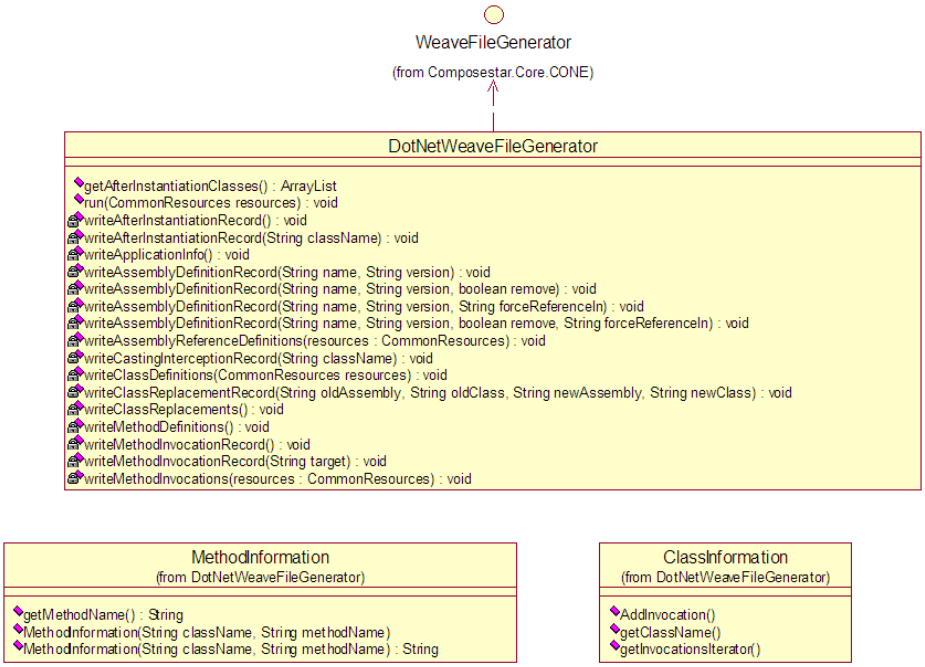


Figure 9.1: Class diagram DotNETWeaveFileGenerator, Compose* module CONE.

be referenced. This means all missing assemblies have to be added. The reason for this is to provide the runtime interpreter with a way to work with all the types in the application, e.g. to create internals.

The method definitions:
The method definitions are forming the interface to the *MessageHandlingFacility* and *CastingFacility* of the runtime interpreter. For every method referenced, a method signature has to be defined in the method definitions block. The signature is linked to a shorthand name, which has to be used in the rest of the definition blocks in the weave specification file.
The implementation of the *MessageHandlingFacility* and the *CastingFacility* can be found respectively in
Composestar.RuntimeDotNET.FLIRT.DotNETMessageHandlingFacility and
Composestar.RuntimeCore.FLIRT.CastingFacility.

The application info:
Because the runtime interpreter has to be initialised, we make use of the application block. The *MessageHandlingFacility* of the runtime interpreter contains a special method, *handleApplicationStart*, to initialise the interpreter. The shorthand name to reference this method is created in the method definitions block.

The class definitions:
The information needed to construct the class definitions is gathered from the Compose* repository in a few steps.
The first step is finding and writing out the method invocations, cast interceptions, and class replacements that have to be applied everywhere in the target program. To apply these

changes everywhere in the target program, we make use of the wildcard as class name, *<class name="*">*. The method invocations are gathered and written out in the function *writeMethod-Invocations*. Basically this function adds a method invocation for every method of a class that is found as a concern, i.e. using a wildcard as method name. After that, the cast interceptions are collected and written out in the *writeCastingInterceptions* function. In Compose* the cast interceptions consist of all classes used as internals in a concern. As last part of the first step the class replacements are written out, i.e. if a certain classname has to be replaced by another classname it is defined here. In Compose* this is done to remove all the references to the *dummy* library used during the compilation phase of the target program.
The second step is writing out the class definitions for concerns that have outputfilters defined. This is done by iterating over all concerns and checking if they have outputfilters. Once we find a concern with outputfilters, we write out the class definition for that concern. This class definition includes the definition for the instantiation of the class, and any method invocation needed to enforce the defined outputfilters on this class.
The third, and last, step is writing out the class definitions for concerns that only require class instantiation notification. For these concerns, the runtime interpreter only has to be notified when an instance of the class implementing the concern is created in the runtime environment. For example classes that are used as *external* in other concerns.

9.2 Invoking the weaver

As the CIL Weaving Tool is a stand-alone program it has to be run or executed by the AOP solution. In the case of Compose*m the ILICIT module is responsible for running the CIL Weaving Tool. To do this, ILICIT has to collect the following information: verification status, debug context, weave specification file, and the target assemblies.
The verification switch is included in the configuration file for the Compose* project. Based on the debug status of the Compose* project itself the debug status for the CIL Weaving Tool is decided. The name of the weave specification file created by CONE is supplied as an argument to the CIL Weaving Tool, so it can find the weave specifications. Finally, all the assemblies that have to be woven are supplied as argument to the CIL Weaving Tool. To be able to handle large projects, Compose* has the option to supply a file to the CIL Weaving Tool containing the names of all the assemblies that have to be woven. However this is only done if the number of files is twenty or more, to prevent any problems with commandline prompt overflows. ILICIT will create this file and supply the filename as argument to to CIL Weaving Tool.

Figure 9.2 shows the class diagram for the ILICIT module. The *main* and *run* functions are required by the Compose* module framework and are the entry points of the ILICIT module. The four functions *castingInterceptions*, *getAfterInstantiationClasses*, *getConcernsWithFMO*, and *getConcernsWithOutputFilters* are for constructing the target assemblies list. Each function checks a different AOP construction for concerns which require weaving. From this list of concerns a list of assemblies is created, by finding the assemblies in which the concerns are defined. This list of assembly names is provided to the CIL Weaving Tool.

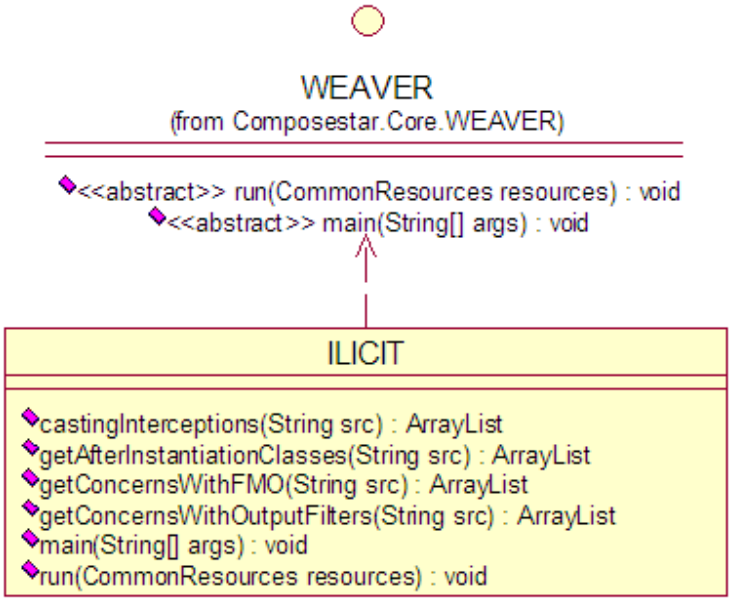


Figure 9.2: Class diagram ILICIT.

9.3 Summary

This chapter described the integration of the CIL Weaving Tool in the Compose* architecture. The integration consists of two steps: creating the weave specifications for the CIL Weaver Tool, and invoking the CIL Weaver Tool. Figure 9.3 shows an overview of the Compose* architecture and highlights the parts used in the integration of the CIL Weaving Tool. Responsible for writing out the repository and weave specification is the CONE module. The weave specification, together with the assemblies compiled by the Compose* compile time are supplied to the CIL Weaving Tool as input. Invoking the CIL Weaving Tool with the right arguments is the task of ILICIT. The repository and the modified assemblies (the output from the CIL Weaving Tool), together with the Compose* runtime allow for the execution of the aspects when the program runs.

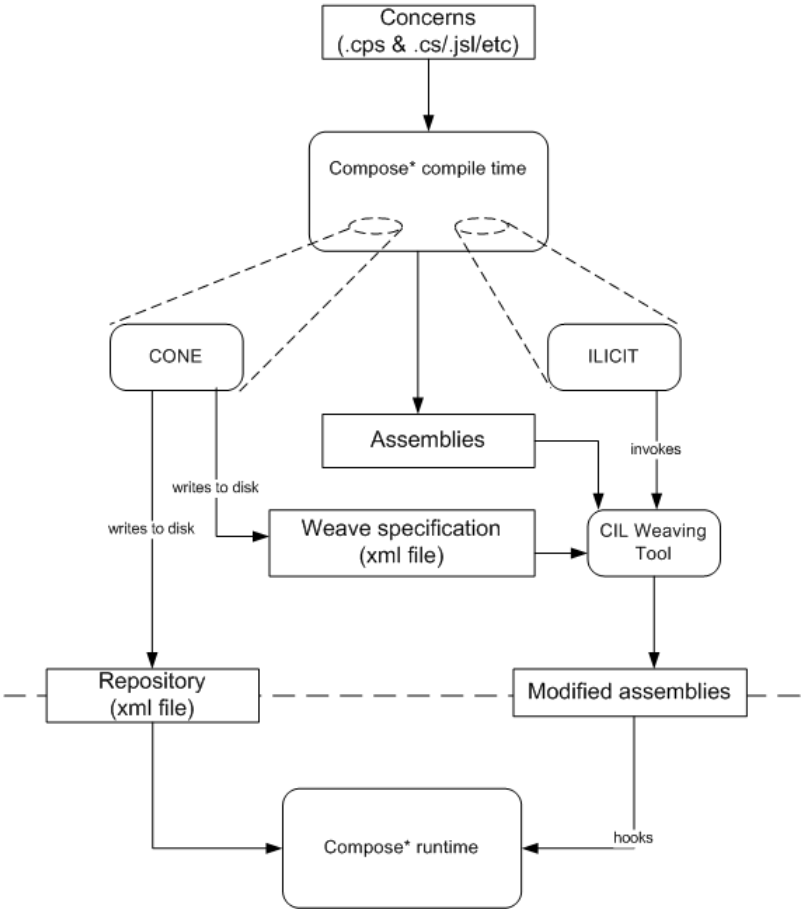


Figure 9.3: The integration of the CIL Weaver Tool in the Compose* architecture.

CHAPTER 10

Conclusion and future work

To express crosscutting concerns in a clear manner, the aspect-oriented programming (AOP) paradigm was introduced. In AOP languages, crosscutting concerns are defined in *aspects*. These aspects are composed, or woven, with *base components*. These components encapsulate functionality expressed in traditional object-oriented languages. By separating the crosscutting concerns from the components, the AOP paradigm tries to solve problems as code tangling and code scattering.

As the aop language and base component language can be separated, an AOP solution can be implemented independent of the base component language. In other words, the languages used to define crosscutting locations can be different from the language used to define the base components. A suitable platform for such an AOP solution is the .NET Framework, since, in principle, this platform can support a wide range of different component languages (i.e. programming languages). The .NET platform uses an intermediate language, the Common Intermediate Language (CIL), to which all the programming languages are mapped at compile time. At runtime the CIL code is compiled into native machine language by the Just-in-Time (JIT) compiler and executed. To distribute applications, the CIL code is wrapped into binary files, called assemblies.

To weave aspects and components together, we have identified four possible approaches.
The first approach is source code weaving. The idea is to weave the aspect source and component source together before compiling it to the CIL, using the native language compiler.
The second approach uses the profiling API's to weave at the JIT compiling phase. The profiling API's offer hooks into the .NET runtime machine, the Common Language Runtime (CLR). These hooks allow the change of the CIL code just before it is compiled into native machine language.
The third approach is the adaptation of the CLR, the .NET runtime machine. The logic to decide where and when to execute certain aspects can be built into the CLR.
The fourth, and last, approach is weaving the aspects into the .NET assemblies containing the component code. Both the advice code and base component code are compiled using the native language compilers. Afterwards the assemblies, containing the component code, are modified

to include hooks to the aspect code. These hooks can be directly linked to the aspect code or they can be linked to an interpreter, which in turn will call the appropriate aspect code to execute.

Based on the advantages and disadvantages of each approach, we have decided to create a compile time CIL weaver (third approach). The purpose of this CIL Weaving Tool is to make transformations to the .NET Common Intermediate Language (CIL) code available to other tools. An example of a tool, that can use the weaver tool, is an AOP implementation for the .NET Framework.

This CIL Weaving Tool consists of two parts: the *PE Weaver* and the *IL Weaver*.
The PE Weaver extracts the IL code from the assemblies, which are binary files, and creates the corresponding textual IL files. This extraction process is called *disassembling*, and is done with the help of the *ildasm tool* (IL Disassembler), distributed with the .NET SDK. These textual IL files, together with the *weave specification file*, form the input for the IL Weaver. After the IL Weaver has updated the textual IL files, according to the information supplied in the weave specification file, they are transformed into assemblies again by the PE Weaver. This process is called *assembling*, and is done with the *ilasm tool* (IL Assembler), distributed with the standard .NET installation.
The IL Weaver is responsible for actual weaving of the instructions contained in the weave specification file. Internally the IL Weaver consists of five processes: a reader for the weave specification file (also creates an internal representation of it), a reader for the textual IL file (also makes an internal representation of it), the weaver process, and a writer to write out the modified textual IL file.
This *weave specification file* is an xml file to specify the set of *weave operations*, which can be based on AOP constructs. The general structure of the weave specification file consists of the following blocks: the assembly reference block (references to other assemblies to include/exclude), the method definition block (defines signatures of methods which can be referenced in the other blocks), the application block (defines modifications at application level), and the class block (defines modifications at class level, a class block is identified by its fully qualified name).

10.1 Future work on the CIL Weaving Tool

Future work on the CIL Weaving Tool can involve the following issues:

Improve performance of the reading and writing of assemblies: The current approach, used to extract the CIL code from the assemblies, is to disassemble them using the .NET disassemble tool *ildasm*. After wards the CIL Weaving Tool performs weaving operations on the textual IL code. Finally, the .NET assemble tool *ilasm* is used to create the new assembly. This process however relies heavily on I/O operations, and therefore can become slow. A way to improve performance is to extract the CIL code from the assembly directly into the internal structure, representing the CIL code, used by the weaver. Doing so, partly eliminates the slow down caused by the processing of textual IL files, which can become very large compared to the size of the assembly.

Expand the internal structure representing the CIL code: In its current state the CIL Weaving Tool uses a very limited internal structure to represent all the CIL code structures. Only information needed for the weaving process is fully parsed and placed in an internal AST, the rest of the information is just stored without parsing. Allthough this approach

allows a certain amount of changes to the CIL specifications without affecting the weaver, it can also limit the implementation of new weave points. This is due to the fact that the information needed for the weave point may not be parsed. To have all the information available, the CIL code has to be fully parsed and stored in an internal AST.

Increase the number of possible weave points: The weave points implemented in the current version are the most basic weave points possible. A lot of additional weave points can be added to support the most advanced AOP constructs. For example weave points related to the try and catch block can be added.

Refine the selection of existing weave points: Currently implemented weave points can be refined. For example the weave points identified by constructors/methods are currently only matched by name, in addition they could also be matched by signature.

Update the Program Debug Database (PDB) file: By modifying the CIL code, any information stored in the PDB files for debugging the program is rendered invalid. But it might be possible to update the information in the PDB files according to the modifications made to the CIL code. This would allow the developer to use native debugging tools during the development phase of a program.

Bibliography

[1] Ada. Ada for the web, 1996. URL http://www.acm.org/sigada/wg/web_ada/.

[2] aspxreme. Getting Started with ASP.NET. Technical report, aspxtreme, 2004. URL http://authors.aspalliance.com/aspxtreme/aspnet/index.aspx.

[3] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994. URL http://trese.cs.utwente.nl/publications/paperinfo/bergmans.phd.pi.top.htm.

[4] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[5] S. R. Boschman. Performing transformations on .NET intermediate language code. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[6] R. Bosman. Automated reasoning about Composition Filters. Master's thesis, University of Twente, The Netherlands, Nov. 2004.

[7] Castle Project. Aspect#. Technical report, 2005. URL http://www.castleproject.org/index.php/AspectSharp.

[8] S. Chiba and M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, pages 364–376, 2003.

[9] O. Conradi. Fine-grained join point model in Compose*. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[10] D. Doornenbal. Analysis and redesign of the Compose* language. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[11] P. E. A. Dürr. Detecting semantic conflicts between aspects (in Compose*). Master's thesis, University of Twente, The Netherlands, Apr. 2004.

[12] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Comm. ACM*, 44(10): 29–32, Oct. 2001.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.

[14] C. F. N. García. Compose* - a runtime for the .NET platform. Master's thesis, Vrije Universiteit Brussel, Belgium, Aug. 2003.

[15] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, 1995. URL `http://trese.cs.utwente.nl/publications/paperinfo/glandrup.thesis.pi.top.htm`.

[16] M. A. Gnter Kniesel, Pascal Costanza. Jmangler - a powerful back-end for aspect-oriented programming. In T. E. R. Filman and M. A. S. Clarke, editors, *Aspect-oriented Software Development*. Prentice Hall, 2004. URL `http://roots.iai.uni-bonn.de/research/jmangler/downloads/papers/kniesel2003_aosdBook.pdf`. To appear.

[17] J. Gough. *Compling for the .NET Common Language Runtime (CLR)*. Prentice Hall, 2001. ISBN 0-13-062296-6.

[18] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, 2003. ISBN 0471431044.

[19] W. Havinga. Designating join points in Compose* - a predicate-based superimposition language for Compose*. Master's thesis, University of Twente, The Netherlands, May 2005.

[20] F. J. B. Holljen. Compilation and type-safety in the Compose* .NET environment. Master's thesis, University of Twente, The Netherlands, May 2004.

[21] R. L. R. Huisman. Debugging Composition Filters. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[22] S. H. G. Huttenhuis. Patterns within aspect orientation. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[23] E. International. Common language infrastructure (CLI). Standard ECMA-335, ECMA International, 2002. URL `http://www.ecma-international.org/publications/files/ecma-st/Ecma-335.pdf`.

[24] A. Jackson and S. Clarke. SourceWeave.NET: Cross-Language Aspect-Oriented Programming. Technical report, Trinity College Dublin, 2004.

[25] Jython. Jython homepage. URL `http://www.jython.org/`.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[27] P. Koopmans. Sina user's guide and reference manual. Technical report, Dept. of Computer Science, University of Twente, 1995. URL `http://trese.cs.utwente.nl/publications/paperinfo/sinaUserguide.pi.top.htm`.

[28] D. Lafferty and V. Cahill. Language-Independent Aspect-Oriented Programming. Technical report, 2003.

[29] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification (2nd edition)*. Addison-Wesley Pub Co, 1999. ISBN 0201432943.

[30] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, pages 105–121, nov 2003.

[31] Microsoft Corporation. Profiling. Technical report, Microsoft Corporation, 2002.

[32] Microsoft Corporation. Overview of the .NET framework. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrintroductiontonetframeworksdk.asp`.

[33] Microsoft Corporation. What is the common language specification. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconwhatiscommonlanguagespecification.asp`.

[34] Microsoft Corporation. .NET compact framework - technology overview. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/overview/default.aspx`.

[35] Microsoft Corporation. What's is .NET? Technical report, Microsoft Corporation, 2005. URL `http://www.microsoft.com/net/basics.mspx`.

[36] Microsoft Corporation. "Assemblies". Technical report, Microsoft Corporation, 2004. URL `http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassemblies.asp`.

[37] Microsoft Corporation. "Arrays". Technical report, Microsoft Corporation, 2004.

[38] Microsoft Corporation. "Common Type System". Technical report, Microsoft Corporation, 2004. URL `http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthecommontypesystem.asp`.

[39] Microsoft Corporation. Programming with Assemblies. Technical report, Microsoft Corporation, 2004. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconprogrammingwithassemblies.asp`.

[40] Microsoft Corporation. Phoenix Framework. Technical report, 2005. URL `http://research.microsoft.com/phoenix/technical.aspx`.

[41] Mono. Mono. URL `http://www.mono-project.com`.

[42] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, The Netherlands, June 2006.

[43] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In M. Akşit, editor, *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2001. ISBN 0-7923-7576-9.

[44] M. Pietrek. An in-depth look into the win32 portable executable file format. *MSDN Magazine*, February 2002.

[45] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, Apr. 2002.

[46] A. Popovici, G. Alonso, and T. Gross. Just in time aspects. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 100–109. ACM Press, Mar. 2003.

[47] J. Prosise. *Programming Microsoft .NET*. Microsoft Press, Redmond, WA, USA, 2002. ISBN 0-7356-1376-1.

[48] Rotor. ROTOR. URL `http://msdn.microsoft.com/net/sscli/`.

[49] P. Salinas. Adding systemic crosscutting and super-imposition to Composition Filters. Master's thesis, Vrije Universiteit Brussel, Aug. 2001.

[50] W. Schult, P. Troeger, and A. Polze. LOOM .NET - An Aspect Weaving Tool. Technical report, 2003.

[51] D. Shukla, S. Fell, and C. Sells. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. *MSDN Magazine*, March 2002.

[52] D. R. Spenkelink. Compose* incremental. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[53] T. Staijen. Towards safe advice: Semantic analysis of advice types in Compose*. Master's thesis, University of Twente, Apr. 2005.

[54] D. Stutz. The Microsoft shared source CLI implementation. 2002.

[55] P. Tarr, H. Ossher, S. M. Sutton, Jr., and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 37–61. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[56] J. W. te Winkel. Bringing Composition Filters to C. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[57] The Apache Jakarta Project. The Byte Code Engineering Library. Technical report, 2005. URL `http://jakarta.apache.org/bcel/`.

[58] M. D. W. van Oudheusden. Automatic derivation of semantic properties in .NET. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[59] C. Vinkes. Superimposition in the Composition Filters model. Master's thesis, University of Twente, The Netherlands, Oct. 2004.

[60] D. Watkins. Handling language interoperability with the Microsoft .NET framework. Technical report, Monash Univeristy, Oct. 2000. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/interopdotnet.asp`.

[61] D. Watkins, M. Hammond, and B. Abrams. *Programming in the .NET Environment*. Addison Wesley Professional, 2002. ISBN 0201770180.

[62] D. A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.

[63] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente, 1999. URL `http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.top.htm`.

APPENDIX A

The CIL Instruction Set

The following tables contain a brief description of each instruction of the *Common Intermediate Language* instruction set, the instructions are grouped by the type of their argument. The tables are taken from *Compiling for the .NET Common Language Runtime (CLR)* [17] and are included for reference purposes only.

The final column in the tables give the *stack*-Δ for the instruction. Instructions marked with " *" have a *stack*-Δ that depends on the arguments of the call. In the case of instruction prefixes, the *stack*-Δ depends on the following instruction. [17]

Instruction	**Description**	Δ
add	Add two top elements on stack	-1
add.ovf	Add two top elements on stack and trap signed overflow	-1
add.ovf.un	Add two top elements on stack and trap unsigned overflow	-1
and	Bitwise *and* of two top-of-stack elements	-1
arglist	Push the arglist of a "varargs" method	1
break	Breakpoint instruction for debugging	0
ceq	Compare for equality and push bool	-1
cgt	Compare for greater than and push bool	-1
cgt.un	Compare for greater than unsigned and push bool	-1
ckfinite	Throw an exception if top of stack is a *NaNS*	0
clt	Compare for less than and push bool	-1
clt.un	Compare for less than unsigned and push bool	-1
conv.i	Convert top of stack to natural integer	0
conv.i1	Convert top of stack to 1-byte integer	0
conv.i2	Convert top of stack to 2-byte integer	0
conv.i4	Convert top of stack to 4-byte integer	0
conv.i8	Convert top of stack to 8-byte integer	0
conv.ovf.i	Convert top of stack to natural integer and trap overflow	0

Instruction	Description	Δ
conv.ovf.i.un	Convert from unsigned on top of stack to natural integer and trap overflow	0
conv.ovf.i1	Convert top of stack to int8 and trap overflow	0
conv.ovf.i1.un	Convert from unsigned on top of stack to int8 and trap overflow	0
conv.ovf.i2	Convert top of stack to int16 and trap overflow	0
conv.ovf.i2.un	Convert from unsigned on top of stack to int16 and trap overflow	0
conv.ovf.i4	Convert top of stack to int32 and trap overflow	0
conv.ovf.i4.un	Convert from unsigned on top of stack to int32 and trap overflow	0
conv.ovf.i8	Convert top of stack to int64 and trap overflow	0
conv.ovf.i8.un	Convert from unsigned on top of stack to int64 and trap overflow	0
conv.ovf.u	Convert top of stack to natural uint and trap overflow	0
conv.ovf.u.un	Convert from unsigned on top of stack to natural uint and trap overflow	0
conv.ovf.u1	Convert top of stack to natural uint8 and trap overflow	0
conv.ovf.u1.un	Convert from unsigned on top of stack to natural uint8 and trap overflow	0
conv.ovf.u2	Convert top of stack to natural uint16 and trap overflow	0
conv.ovf.u2.un	Convert from unsigned on top of stack to natural uint16 and trap overflow	0
conv.ovf.u4	Convert top of stack to natural uint32 and trap overflow	0
conv.ovf.u4.un	Convert from unsigned on top of stack to natural uint32 and trap overflow	0
conv.ovf.u8	Convert top of stack to natural uint64 and trap overflow	0
conv.ovf.u8.un	Convert from unsigned on top of stack to natural uint64 and trap overflow	0
conv.r4	Convert top-of-stack value to float32	0
conv.r8	Convert top-of-stack value to float64	0
conv.u	Convert top-of-stack value to natural uint	0
conv.u1	Convert top-of-stack value to uint8	0
conv.u2	Convert top-of-stack value to natural uint16	0
conv.u4	Convert top-of-stack value to uint32	0
conv.u8	Convert top-of-stack value to uint64	0
cpblk	Support for ANSI C ”memcopy” function	-3
div	Signed division of two top elements on stack	-1
div.un	Unsigned division of two top elements on stack	-1
dup	Duplicate the top-of-stack value	1
endfilter	Return value from filter block	-1
endfinally	Return from finally block	0
initblk	Support for ANSI C ”memset” function	-3
ldarg.0	Load the zeroth argument	1
ldarg.1	Load the first argument	1
ldarg.2	Load the second argument	1
ldarg.3	Load the third argument	1
ldc.i4.0	Load literal 0 of type int32	1

Instruction	Description	Δ
ldc.i4.1	Load literal 1 of type int32	1
ldc.i4.2	Load literal 2 of type int32	1
ldc.i4.3	Load literal 3 of type int32	1
ldc.i4.4	Load literal 4 of type int32	1
ldc.i4.5	Load literal 5 of type int32	1
ldc.i4.6	Load literal 6 of type int32	1
ldc.i4.7	Load literal 7 of type int32	1
ldc.i4.8	Load literal 8 of type int32	1
ldc.i4.M1	Load literal -1 of type int32	1
ldelem.i	Load natural integer array element	1
ldelem.i1	Load int8 array element	1
ldelem.i2	Load int16 array element	1
ldelem.i4	Load int32 array element	1
ldelem.i8	Load int64 array element	1
ldelem.r4	Load float32 array element	1
ldelem.r8	Load float64 array element	1
ldelem.ref	Load reference array element	1
ldelem.u	Load natural unsigned integer array element	1
ldelem.u1	Load uint8 array element	1
ldelem.u2	Load uint16 array element	1
ldelem.u4	Load uint32 array element	1
ldind.i	Load natural integer pointed to by top of stack	0
ldind.i1	Load int8 pointed to by top of stack	0
ldind.i2	Load int16 pointed to by top of stack	0
ldind.i4	Load int32 pointed to by top of stack	0
ldind.i8	Load int64 pointed to by top of stack	0
ldind.r4	Load float32 pointed to by top of stack	0
ldind.r8	Load float64 pointed to by top of stack	0
ldind.ref	Load reference pointed to by top of stack	0
ldind.u	Load natural unsigned integer pointed to by top of stack	0
ldind.u1	Load uint8 pointed to by top of stack	0
ldind.u2	Load uint16 pointed to by top of stack	0
ldind.u4	Load uint32 pointed to by top of stack	0
ldlen	Load length of array referenced by top of stack	0
ldloc.0	Load zeroth local variable	1
ldloc.1	Load first local variable	1
ldloc.2	Load second local variable	1
ldloc.3	Load third local variable	1
ldnull	Load a *null* value on the stack	1
localloc	Expand the current activation record	-1
mul	Multiply top-of-stack elements	1
mul.ovf	Multiply top-of-stack elements and trap overflow	1
mul.ovf.un	Multiply top-of-stack elements and trap unsigned overflow	1
neg	Arithmetically negate top-of-stack element	0
nop	Do nothing	0

Instruction	Description	Δ
not	Bitwise *negate* top-of-stack element	0
or	Bitwise *or* top-of-stack elements	-1
pop	Discard top-of-stack element	-1
refanytype	Extract type token from typed reference	0
rem	Remainder of two top-of-stack elements	-1
rem.un	Unsigned remainder of two top-of-stack elements	-1
ret	Return to caller, maybe with function result	*
rethrow	Rethrow the current exception	0
shl	Arithmetic shift left	-1
shr	Arithmetic shift right	-1
shr.un	Logical shift right	-1
stelem.i	Store array element of natural integer type	-3
stelem.i1	Store array element of int8 type	-3
stelem.i2	Store array element of int16 type	-3
stelem.i4	Store array element of int32 type	-3
stelem.i8	Store array element of int64 type	-3
stelem.r4	Store array element of float32 type	-3
stelem.r8	Store array element of float64 type	-3
stelem.ref	Store array element of reference type	-3
stind.i	Store top of stack to natural integer pointer target	-2
stind.i1	Store top of stack to int8 pointer target	-2
stind.i2	Store top of stack to int16 pointer target	-2
stind.i4	Store top of stack to int32 pointer target	-2
stind.i8	Store top of stack to int64 pointer target	-2
stind.r4	Store top of stack to float32 pointer target	-2
stind.r8	Store top of stack to float64 pointer target	-2
stind.ref	Store top of stack reference to pointer target	-2
stloc.0	Store top of stack to zeroth local variable	-1
stloc.1	Store top of stack to first local variable	-1
stloc.2	Store top of stack to second local variable	-1
stloc.3	Store top of stack to third local variable	-1
sub	Subtract top-of-stack elements	-1
sub.ovf	Subtract top-of-stack elements and trap overflow	-1
sub.ovf.un	Subtract top-of-stack elements and trap unsigned overflow	-1
tail.	Prefix. Following call terminates method	*
throw	Throw top-of-stack object as exception	-1
unaligned.	Prefix. Pointer on top of stack may be unaligned	*
volatile.	Prefix. Address on top of stack is of volatile location	*
xor	Bitwise *xor* of top-of-stacks elements	-1

Table A.1: Instructions with no arguments

Instruction		Description	Δ
ldarg	Num	Load the N-th argument	1
ldarg.s	Num	Load the N-th argument (short form)	1
ldarga	Num	Load the N-th argument address	1
ldarga.s	Num	Load the N-th argument address (short form)	1
starg	Num	Store the N-th argument	-1
starg.s	Num	Store the N-th argument (short form)	-1
ldloc	Num	Load the N-th local variable	1
ldloc.s	Num	Load the N-th local variable (short form)	1
ldloca	Num	Load address of the N-th local variable	1
ldloca.s	Num	Load address of the N-th local variable (short form)	1
stloc	Num	Store top of stack to N-th local variable	-1
stloc.s	Num	Store top of stack to N-th local variable (short form)	-1
ldc.i4	Num	Load literal N as int32	1
ldc.i4.s	Num	Load literal N as int32 (short form)	1
ldc.i8	Num	Load literal N as int64	1
ldc.r4	Num	Load literal N as float32	1
ldc.r8	Num	Load literal N as float64	1

Table A.2: Instructions with a numeric argument

Instruction		Description	Δ
box	TRef	Create boxed copy of top-of-stack value of type T	0
castclass	TRef	Cast top-of-stack reference to type T	0
cpobj	TRef	Copy value object of type T	-2
initobj	TRef	Initialize value of type T	-1
isinst	TRef	Test if top of stack is an instance of type T	0
ldelema	TRef	Load address of array element of type T	-1
ldobj	TRef	Load value of type T onto stack	0
mkrefany	TRef	Make typed reference of type T from top-of-stack pointer	0
newarr	TRef	Create array of element type T	0
refanyval	TRef	Extract pointer from typed reference of type T	0
sizeof	TRef	Load size in bytes of value type T	1
stobj	TRef	Store top-of-stack value of type T	2
unbox	TRef	Create managed pointer to boxed value	0

Table A.3: Instructions with a type reference argument

Instruction		Description	Δ
beq	Lab	Branch to label if equal	-2
beq.s	Lab	Branch to label if equal (short form)	-2
bge	Lab	Branch to label if greater or equal	-2
bge.s	Lab	Branch to label if greater or equal (short form)	-2
bge.un	Lab	Branch to label if unsigned greater or equal	-2
bge.un.s	Lab	Branch to label if unsigned greater or equal (short form)	-2
bgt	Lab	Branch to label if greater than	-2
bgt.s	Lab	Branch to label if greater than (short form)	-2
bgt.un	Lab	Branch to label if unsigned greater than	-2
bgt.un.s	Lab	Branch to label if unsigned greater than (short form)	-2
ble	Lab	Branch to label if less than or equal	-2
ble.s	Lab	Branch to label if less than or equal (short form)	-2
ble.un	Lab	Branch to label if unsigned less than or equal	-2
ble.un.s	Lab	Branch to label if unsigned less than or equal (short form)	-2
blt	Lab	Branch to label if less than	-2
blt.s	Lab	Branch to label if less than (short form)	-2
blt.un	Lab	Branch to label if unsigned less than	-2
blt.un.s	Lab	Branch to label if unsigned less than (short form)	-2
bne.un	Lab	Branch to label if unequal or unordered	-2
bne.un.s	Lab	Branch to label if unequal or unordered (short form)	-2
br	Lab	Unconditional branch to label	0
br.s	Lab	Unconditional branch to label (short form)	0
brfalse	Lab	Branch to label if top of stack zero or null	-1
brfalse.s	Lab	Branch to label if top of stack zero or null (short form)	-1
brtrue	Lab	Branch to label if top of stack not zero or null	-1
brtrue.s	Lab	Branch to label if top of stack not zero or null (short form)	-1
leave	Lab	Exit from try, catch, or filter block	0
leave.s	Lab	Exit from try, catch, or filter block (short form)	0

Table A.4: Instructions with a label argument

Instruction		Description	Δ
call	MRef	Statically call specified method	*
callvirt	MRef	Virtual call of specified method	*
jmp	MRef	Jump from current method to MRef	0
ldftn	MRef	Load function pointer to specified method	1
ldvirtftn	MRef	Load virtual function pointer of top-of-stack object	0
newobj	MRef	Allocate new object and call constructor	*

Table A.5: Instructions with a method reference argument

Instruction		Description	Δ
ldfld	FRef	Load field of object with reference on the top of stack	0
ldflda	FRef	Load address of field of top-of-stack object	0
ldsfld	FRef	Load static field of specified class	1
ldsflda	FRef	Load address of static field of specified class	1
stfld	FRef	Store top-of-stack value to field of object next on stack	-2
stsfld	FRef	Store top of stack to static field of specified class	-1

Table A.6: Instructions with a field reference argument

Instruction		Description	Δ
ldstr	Str	Load literal string	1
calli	Sig.	Indirect method call with specified signature	*
ldtoken	Token	Load runtime handle of metadata token	1
switch	...	Table switch on value	-1

Table A.7: Miscellaneous CIL instructions

APPENDIX B

A HelloWorld example in the CIL

In this appendix an example program written in the Common Intermediate Lanuage (CIL) is given. It is a simple console application printing the words 'Hello world' to the screen. See section 5.5 for a detailed description of the example.

Listing B.1: The infamous HelloWorld example written in the CIL

```
1  .assembly extern mscorlib
2  {
3    .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
4    .ver 1:0:5000:0
5  }
6
7  .assembly HelloWorld
8  {
9   .hash algorithm 0x00008004
10  .ver 1:0:1720:26694
11 }
12
13 .module HelloWorld.exe
14 .imagebase 0x00400000
15 .subsystem 0x00000003
16 .file alignment 4096
17 .corflags 0x00000001
18
19 .namespace HelloWorldExample
20 {
21   .class private auto ansi beforefieldinit HelloWorldMain
22     extends [mscorlib]System.Object
23   {
24   } // end of class HelloWorldMain
25 }
26
27 .namespace HelloWorldExample
28 {
29   .class private auto ansi beforefieldinit HelloWorldMain
30     extends [mscorlib]System.Object
31   {
32     .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
33     {
34       .maxstack 1
35       IL_0000:  ldarg.0
```

```
36      IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
37      IL_0006:  ret
38    } // end of method .ctor
39
40    .method private hidebysig instance void Run() cil managed
41    {
42      .maxstack 2
43      IL_0000:  ldarg.0
44      IL_0001:  ldstr      "Hello world"
45      IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
46      IL_000b:  ret
47    } // end of method Run
48
49    .method private hidebysig static void Main(string[] args) cil managed
50    {
51      .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = (01 00 00 00)
52      .entrypoint
53      .maxstack 1
54      .locals init (
55        [0]class HelloWorldExample.HelloWorldMain
56      )
57      IL_0000:  newobj     instance void PeWeaverTests.HelloWorld::.ctor()
58      IL_0005:  stloc.0
59      IL_0006:  ldloc.0
60      IL_0007:  callvirt   instance void PeWeaverTests.HelloWorld::Run()
61      IL_000c:  ret
62    } // end of method Main
63
64  } // end of class HelloWorldMain
65 }
```

APPENDIX C

The Weave Specification file

```
<?xml version="1.0" encoding="UTF-8"?>
<weaveSpecification version="<weave specification version>">
  <assemblies>
    <assembly name="<assembly name>"
              version="<assembly version>"
              publicKeytoken="<keytoken of the assembly>"
              forceReferenceIn="[<assembly name>/*]"
              remove="[yes/no]"/>
  </assemblies>
  <methods>
    <method id="<id of the method reference>"
            assembly="<assembly the method can be found in>"
            class="<fully qualified name of the class the method belongs to>"
            name="<name of the method>">
      <argument value="" type="[string/int]"/>
      <argument value="%senderobject"/>
      <argument value="%createdobject"/>
      <argument value="%targetobject"/>
      <argument value="%targetmethod"/>
      <argument value="%originalparameters"/>
      <argument value="%casttarget"/>
    </method>
  </methods>
  <application name="<name of the application>">
    <notifyStart id="<reference to a method definition>"/>
  </application>
  <class name="[*/<fully qualified class name>]">
    <afterClassInstantiation>
      <executeMethod id="<reference to a method definition>"/>
    </afterClassInstantiation>
    <methodInvocations>
      <callToMethod class="<fully qualified name of the class>" name="<name of the method>">
        <voidRedirectTo id="<reference to a method definition>"/>
        <returnvalueRedirectTo id="<reference to a method definition>"/>
      </callToMethod>
    </methodInvocations>
    <casts>
      <castTo assembly="<assembly the method can be found in>" class="">
        <executeMethodBefore id="<reference to a method definition>"/>
      </castTo>
    </casts>

```

```
    <classReplacements>
      <classReplacement assembly="<assembly the method can be found in>" class="">
        <replaceWith assembly="<assembly the method can be found in>" class=""/>
      </classReplacement>
    </classReplacements>
    <fieldAccesses>
      <field class="<fully qualified name of the class>" name="<name of the field>">
        <callBefore id="<reference to a method definition>"/>
        <callAfter id="<reference to a method definition>"/>
        <replaceWith id="<reference to a method definition>"/>
      </field>
    </fieldAccesses>
  </class>
</weaveSpecification>
```

APPENDIX D

Class diagrams Weaver

D.1 PE Weaver

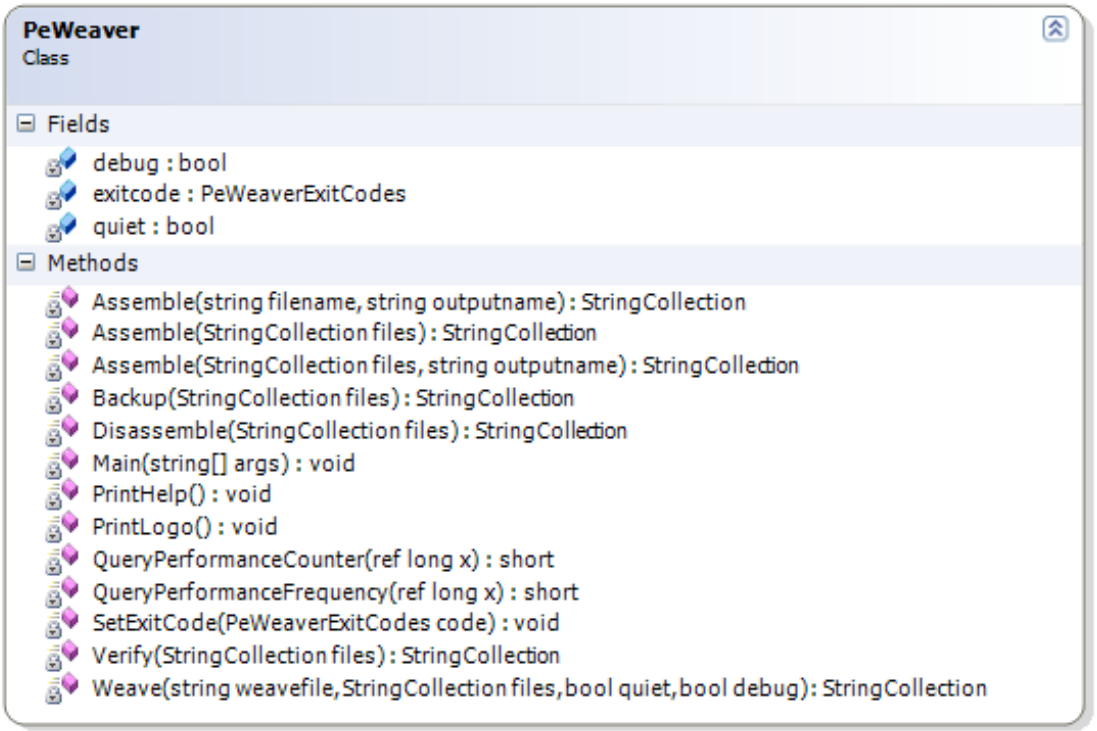


Figure D.1: Class diagram *PeWeaver*.

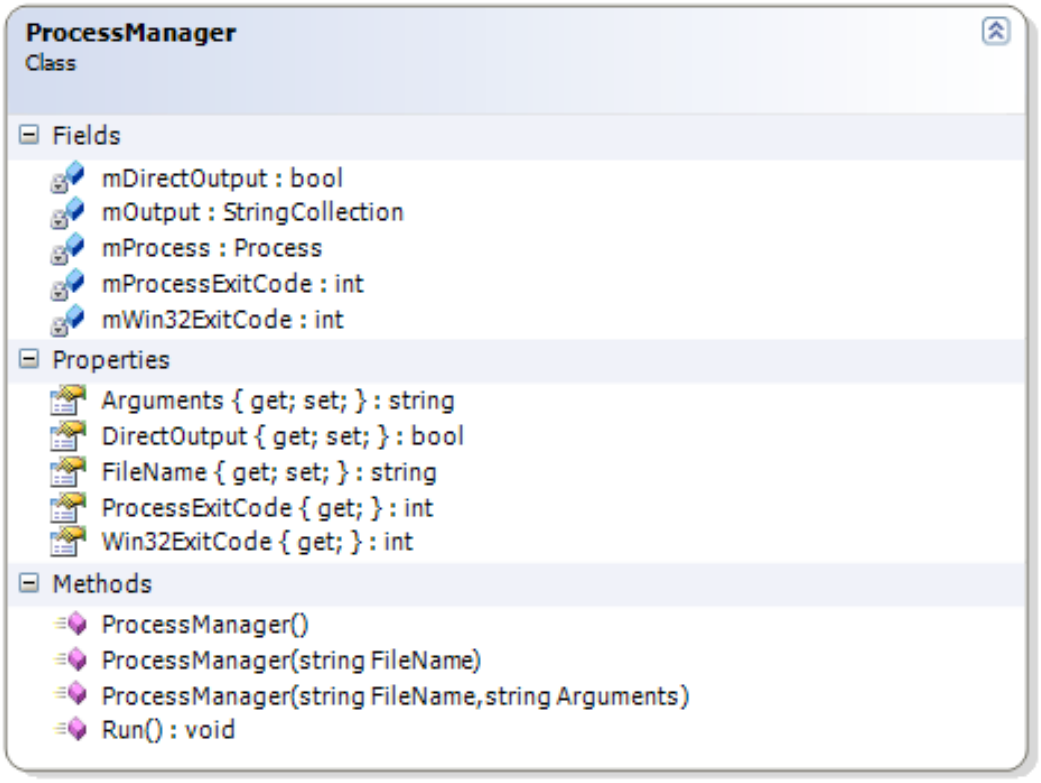


Figure D.2: Class diagram *ProcessManager*.

D.2 IL Weaver

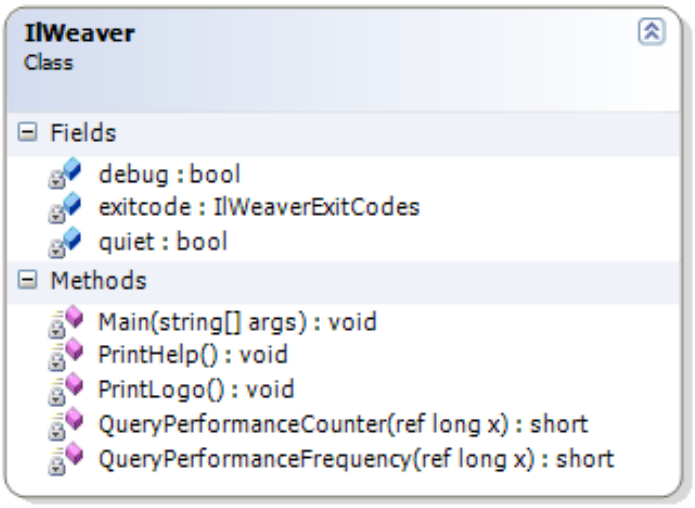
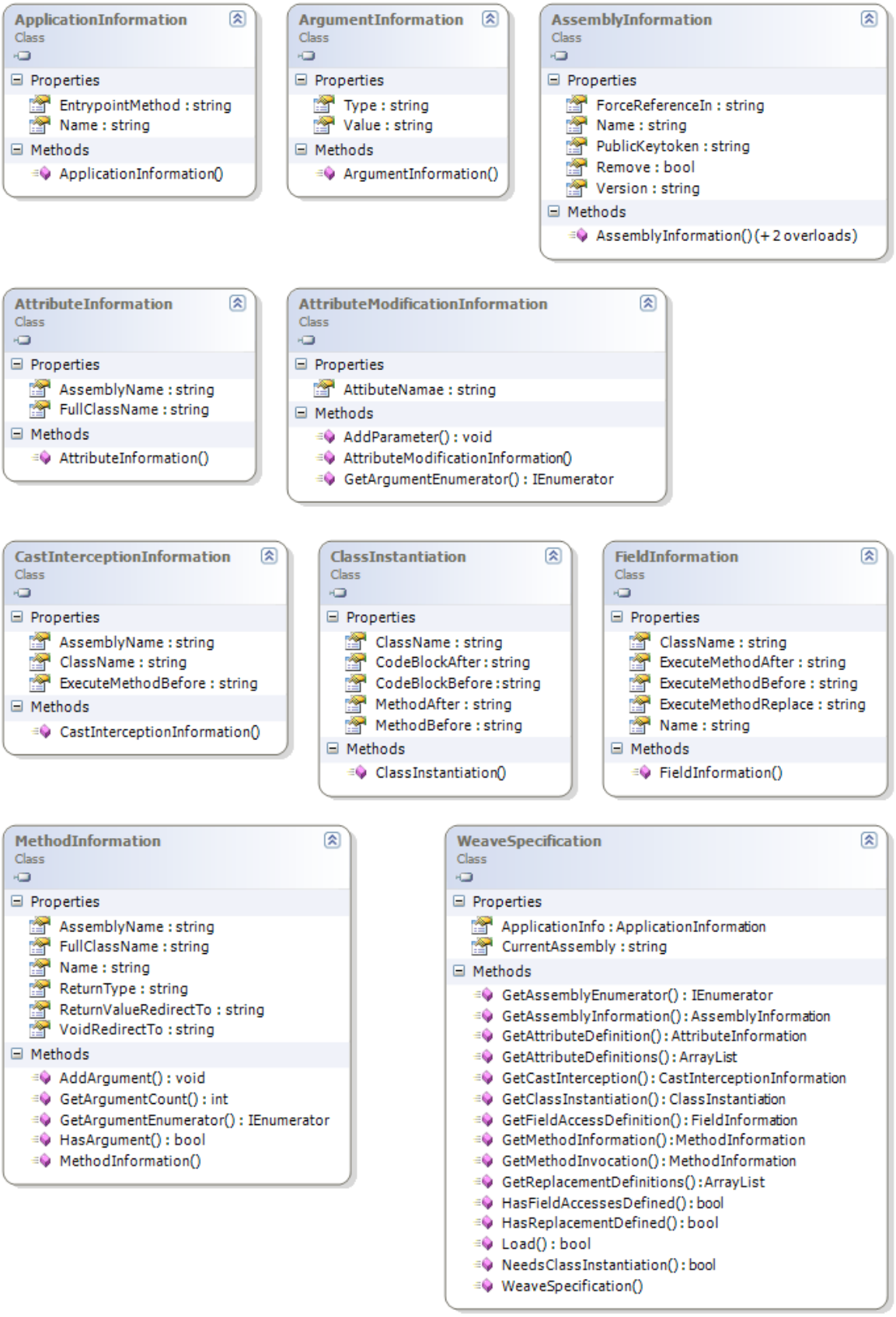


Figure D.3: Class diagram *IL Weaver*.

D.3 WeaveLibrary

Figure D.4: Class diagram internal IL representation, the *IlStructure*.


ApplicationInformation
Class
Properties
EntrypointMethod : string
Name : string
Methods
ApplicationInformation()
ArgumentInformation
Class
Properties
Type : string
Value : string
Methods
ArgumentInformation()
AssemblyInformation
Class
Properties
ForceReferenceIn : string
Name : string
PublicKeytoken : string
Remove : bool
Version : string
Methods
AssemblyInformation() (+ 2 overloads)
AttributeInformation
Class
Properties
AssemblyName : string
FullClassName : string
Methods
AttributeInformation()
AttributeModificationInformation
Class
Properties
AttibuteNamae : string
Methods
AddParameter() : void
AttributeModificationInformation()
GetArgumentEnumerator() : IEnumerator
CastInterceptionInformation
Class
Properties
AssemblyName : string
ClassName : string
ExecuteMethodBefore : string
Methods
CastInterceptionInformation()
ClassInstantiation
Class
Properties
ClassName : string
CodeBlockAfter : string
CodeBlockBefore : string
MethodAfter : string
MethodBefore : string
Methods
ClassInstantiation()
FieldInformation
Class
Properties
ClassName : string
ExecuteMethodAfter : string
ExecuteMethodBefore : string
ExecuteMethodReplace : string
Name : string
Methods
FieldInformation()
MethodInformation
Class
Properties
AssemblyName : string
FullClassName : string
Name : string
ReturnType : string
ReturnValueRedirectTo : string
VoidRedirectTo : string
Methods
AddArgument() : void
GetArgumentCount() : int
GetArgumentEnumerator() : IEnumerator
HasArgument() : bool
MethodInformation()
WeaveSpecification
Class
Properties
ApplicationInfo : ApplicationInformation
CurrentAssembly : string
Methods
GetAssemblyEnumerator() : IEnumerator
GetAssemblyInformation() : AssemblyInformation
GetAttributeDefinition() : AttributeInformation
GetAttributeDefinitions() : ArrayList
GetCastInterception() : CastInterceptionInformation
GetClassInstantiation() : ClassInstantiation
GetFieldAccessDefinition() : FieldInformation
GetMethodInformation() : MethodInformation
GetMethodInvocation() : MethodInformation
GetReplacementDefinitions() : ArrayList
HasFieldAccessesDefined() : bool
HasReplacementDefined() : bool
Load() : bool
NeedsClassInstantiation() : bool
WeaveSpecification()

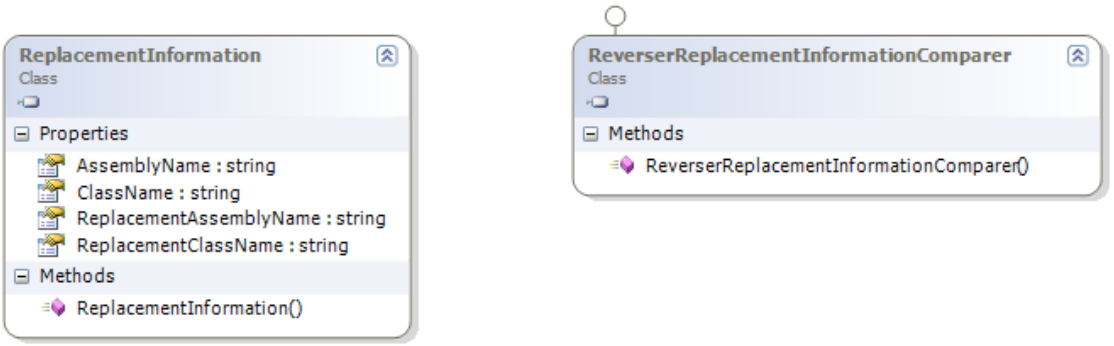


Figure D.5: Class diagram internal representation of the weave specification.

APPENDIX E

Listing DotNETWeaveFileGenerator

Listing E.1: WeaveFileGenerator interface

```
1  package Composestar.Core.CONE;
2
3  import Composestar.Core.Master.CTCommonModule;
4
5  public interface WeaveFileGenerator extends CTCommonModule
6  {
7  }
```

Listing E.2: DotNETWeaveFileGenerator class

```
1  package Composestar.DotNET.CONE;
2
3  /**
4   * This class generates the interception specification file for ILICIT based on
5   * information in the repository.
6   */
7  public class DotNETWeaveFileGenerator implements WeaveFileGenerator
8  {
9    private PrintWriter out = null;
10   private String repository = "repository.xml";
11   private int debugLevel = 0;
12   private String application = "";
13
14   public DotNETWeaveFileGenerator() {
15   }
16
17   private void writeAssemblyReferenceDefinitions( CommonResources resources )
18     throws ModuleException
19   {
20     ...
21   }
22
23   private void writeMethodDefinitions() { ... }
24
25   public ArrayList getAfterInstantiationClasses() { ... }
26
27   private void writeMethodInvocations(CommonResources resources) { ... }
28
29   private void writeClassReplacements() { ... }
```

```
30
31    private void writeClassDefinitions(CommonResources resources) { ... }
32
33    public void run(CommonResources resources) throws ModuleException { ... }
34
35    private void writeApplicationInfo() { ... }
36
37    private void writeAssemblyDefinitionRecord(String name, String version) { ... }
38
39    private void writeAssemblyDefinitionRecord(String name, String version, boolean remove) {
          ... }
40
41    private void writeAssemblyDefinitionRecord(String name, String version,
42                                               String forceReferenceIn) { ... }
43
44    private void writeAssemblyDefinitionRecord(String name, String version, boolean remove,
45                                               String forceReferenceIn) { ... }
46
47    private void writeMethodInvocationRecord() { ... }
48
49    private void writeMethodInvocationRecord(String target) { ... }
50
51    private void writeClassReplacementRecord(String oldAssembly, String oldClass,
52                                             String newAssembly, String newClass) { ... }
53
54    private void writeAfterInstantiationRecord() { ... }
55
56    private void writeAfterInstantiationRecord(String className) { ... }
57
58    private void writeCastingInterceptionRecord(String className) { ... }
59
60
61    class MethodInformation {
62      public MethodInformation(String className, String methodName) { ... }
63
64      public String MethodInformation(String className, String methodName) { ... }
65
66      public String getMethodName() { ... }
67    }
68
69    class ClassInformation {
70      public String getClassName() { ... }
71
72      public void AddInvocation(DotNETWeaveFileGenerator.MethodInformation invocation) { ... }
73
74      public Iterator getInvocationsIterator() { ... }
75    }
76  }
```

APPENDIX F

Listing ILICIT

Listing F.1: WEAVER interface

```
1  package Composestar.Core.WEAVER;
2
3  import Composestar.Core.Exception.ModuleException;
4  import Composestar.Core.Master.CTCommonModule;
5  import Composestar.Core.Master.CommonResources;
6
7  public interface WEAVER extends CTCommonModule {
8    public abstract void run(CommonResources resources) throws ModuleException;
9      public abstract void main(String[] args);
10 }
```

Listing F.2: ILICIT class

```
1  package Composestar.DotNET.ILICIT;
2
3  public class ILICIT implements WEAVER {
4    public void run(CommonResources resources) throws ModuleException {
5      ...
6    }
7
8    /**
9     * @param src Absolute path of a sourcefile
10    * @return ArrayList containing all concerns with FMO and extracted from
11    * the source and its external linked sources
12    */
13   public ArrayList getConcernsWithFMO(String src) {
14     ...
15   }
16
17   /**
18    * @param src Absolute path of a sourcefile
19    * @return ArrayList containing all concerns recognized as a casting interception
20    * Only concerns extracted from the source and its external linked sources are returned
21    */
22   public ArrayList castingInterceptions(String src) throws ModuleException {
23     ...
24   }
25
26   /**
```

```
27      * @param src Absolute path of a sourcefile
28      * @return ArrayList containing all concerns which instantiation should be intercepted
29      * Only concerns extracted from the source and its external linked sources are returned
30      */
31     public ArrayList getAfterInstantiationClasses(String src) throws ModuleException {
32       ...
33     }
34
35     /**
36      * @param src Absolute path of a sourcefile
37      * @return ArrayList containing all concerns with outputfilter(s)
38      * Only concerns extracted from the source and its external linked sources are returned
39      */
40     public ArrayList getConcernsWithOutputFilters(String src) throws ModuleException {
41       ...
42     }
43
44     public void main(String[] args) {
45       ...
46     }
47 }
```