SourceWeave.Net: Cross-Language Source Code Weaving

Andrew Jackson B.Sc.

A dissertation submitted to the University of Dublin,
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

2003

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Andrew Jackson B.Sc.

15thSeptember 2003

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Andrew Jackson B.Sc.

15thSeptember 2003

Acknowledgements

I would like to thank my supervisor, Dr. Siobhán Clarke, for her guidance, insight and interest in this work. Many thanks to Donal Lafferty for all his help and insight into the creation of a weaver and all things .Net. Finally, thank you to the NDS class for making this a year to remember !

Abstract

A well modularised software system reduces complexity and supports change. Separating concerns as a means of achieving good modularisation, is therefore one of the primary principles in software engineering. In general, software development paradigms provide their own specific constructs to support modularisation. Such constructs encapsulate some kinds of concerns and define relationships between them. However, complex entities and relationships exist in many domains that existing software paradigms have lacked the constructs to modularise and express naturally. Concerns are thus forced to cut across each other within the constructs that the paradigm provides. Aspect Oriented Software Development (AOSD) is a relatively new paradigm that enhances concern separation, by providing the means to modularise such concerns, called crosscutting concerns.

Many single language environments have been extended to enable AOSD concepts. Microsoft's .Net platform has opened up the possibility of leveraging AOSD concepts in a multi-language environment. The migration path to AOSD has been established in single language OO platforms, however this is not the case for the .Net multi-language platform. SourceWeave.Net is an aspect-oriented extension to the .Net programming framework. It works as a preprocessor that *weaves* source code, potentially written in different languages, through a rich *joinpoint model*. A joinpoint model determines the extent to which a developer can modularise crosscutting behaviour and compose his code into a functional system. A weaver is a tool that performs composition.

There are many possible approaches to introducing AOSD to .Net. These range from those that work with .Net artifacts, such as source code or IL, to those that weave at different code translation points, such as compile-time, load-time and run-time. SourceWeave.Net's source code manipulation approach has a number of advantages over other approaches. These advantages are in three main areas. Firstly, a programmer can debug the composed system. Secondly, the joinpoint model is rich and highly extensible. Finally, all languages that conform to the .Net framework language standards can be supported.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software systems model real world entities and relationships between entities. The real world, however, can be very difficult to model due to its complexity. Software paradigms provide well-defined constructs that introduce a basis to model entities and relationships. The decomposition of real world problems into distinct concerns allows for the simplification of the problem, with respect to software modelling. However, the separation of these concerns is difficult since, traditional software development paradigms lack the constructs necessary to decompose problems into distinct concerns. Instead, the problem must be forced into the constructs that the paradigm offers, resulting in concerns crosscutting one another. AOSD is a new paradigm that enhances concern separation, through modularisation of crosscutting concerns.

1.2 AOSD

There have been many different approaches to implementing AOSD, such as adaptive programming, aspect-oriented programming, composition filters, hyperspaces, role-modelling and subject-oriented programming. These AOSD implementations have been introduced as extensions to the OO paradigm. OO programming separates concerns in one dimension or through one decomposition construct, the class. AOSD increases the dimensions along which concerns can be separated through enhancement of decomposition constructs [19]. For developers, the separation of concerns means a reduction in code scattering, and code tangling [8].

Code tangling occurs when multiple concerns are implemented within the same modularisation construct. Reuse and maintenance of this module then becomes difficult. The coupling of concerns hampers clean

alteration of one concern without affecting the other. Because of this, re-usability is limited as the module is contextually bound to the system in which it is composed.

Code scattering describes the case where a concern cannot be localised and modularised. Thus the concern is distributed across modularised concerns. This distribution and replication makes both maintenance and change difficult as any alteration to a crosscutting concern will be system wide.

The primary goals of software engineering are to improve software quality, reduce development and maintenance costs and allow for change over time. AOSD is a paradigm that brings software engineering closer to that goal, by reducing code scattering and tangling, and improving modularisation capabilities.

An approach to AOSD is characterised by the joinpoint model it employs. A joinpoint model determines the extent to which a developer can modularise crosscutting behaviour and compose his code into a functional system. There are different conceptual joinpoint models which have been realised in various AOSD implementations. Joinpoint models differ in their ability to modularise and compose concerns. In general, the more extensive the joinpoint model, the more opportunity a developer has to separate and isolate concerns.

1.3 Migration Paths to AOSD

AOSD approaches have been implemented, through paradigm extension, on many single language software development platforms. Microsoft's .Net platform is a relatively new, multi-language platform. Migration paths to AOSD in single language environments have been established. The migration paths of a multi-language environment to AOSD are as yet undefined.

Introduction of AOSD to .Net can be achieved through addition of components to work with .Net artifacts or by modifying existing components within the .Net architecture.

There are three code translation points at which AOSD introduction can be achieved. Firstly, at compile-time through modification of the compilers. Secondly, at load-time through modification of the loader and lastly, at run-time through modification of the run-time environment.

Components can be added that achieve AOSD introduction through modifying .Net of artifacts, such as source code and intermediate language (IL)[1]. To work with source code, a component that works pre-compile-time is required. To work with IL, a component that works pre-run-time is required.

Both modification and extension of the .Net framework have various implications depending on the type of conceptual joinpoint mode employed. Currently, research to assess the implications of each type, and to explore possible migration paths for AOSD approaches to .Net is ongoing.

[1] IL is the .Net equivalent to Java's byte-code

1.4 SourceWeave.Net Contributions

The overall objective of SourceWeave.Net was to implement a cross-language, source code weaver with a rich joinpoint model. Through this implementation we explored source code weaving as a migration path from OO to AOSD framed on the .Net platform. We examined how cross-language weaving and, the implementation of a joinpoint model was achieved and in a multi-language environment through source code manipulation.

1.4.1 Weaving Source Code

One aim of this project was to implement a source code weaver component as an extension of the .Net platform.

SourceWeave.Net extends work done in AspectC#. AspectC# is a tool that weaves modularised cross-cutting concerns with base concerns expressed in C# source code. The aim of AspectC# was to introduce AOSD concepts, at source code level, to C# without language modification.

SourceWeave.Net extends the source code weaving as achieved by AspectC#, to support implementation of a cross-language weaver with a rich joinpoint model.

As discussed in 1.3, there are a number of ways to introduce AOSD to .Net. Much work in the area has been directed toward load-time and run-time, IL-based weavers. Adding a weaving component that works with source code has not been fully investigated as a basis for full implementation of AOSD on the .Net platform. This investigation into the suitability of source code weaving, as a basis for a migration path to AOSD for the .Net platform, is a contribution of this work.

1.4.2 Rich Joinpoint Model

The joinpoint model determines the extent to which concerns can be modularised and composed into a working system. This implementation was aimed at the provision of a rich joinpoint model to allow enhanced separation of concerns (SOC).

The joinpoint models that have been implemented in the AOSD extensions to .Net have generally been limited. Many approaches have focused on how AOSD is implemented rather than focusing on the SOC that can be achieved. As enhanced SOC is the ultimate goal of AOSD, more research should be focused on supporting a rich joinpoint model.

SourceWeave.Net is a proof-of-concept implementation of an extensive joinpoint model at source level in a multi-language environment. This work exposes both the benefits and limitations of implementing such as joinpoint model in this context.

1.4.3 Cross-Language Weaving

Achieving cross-language weaving was a major focus of this work. SourceWeave.Net aimed to leverage the language inter-operability of the .Net platform in the implementation of a cross-language weaver.

There are various approaches which implement language-independent weaving at load-time or run-time. There has been no attempt at weaving modularised crosscutting concerns into base concerns where the expression of these concerns is in source code and is differentiated by language. SourceWeave.Net is a contribution, in that it demonstrates that cross-language weaving can be implemented through source code manipulation.

1.5 Thesis Outline

Chapter 1 In this chapter the we say that paradigms lack the power to capture the complexity of the real world in a natural way. We say that the AOSD paradigm enhances the power to separate concerns and thus model the real world in a natural way.

We then note that AOSD has been established on single language platforms and explain that this is not the case for multi-language platforms. The possible approaches that can be taken to implement AOSD in a multi-language environment are presented and then our approach is introduced.

Finally we present the contributions of this work to the field.

Chapter 2

In this chapter we look at the .Net framework, what it is and the .Net framework architecture. We explored how this architecture could be altered or extended to introduce the AOSD into the framework.

We then examine how differing models of AOSD have been introduced on the Java platform. We then look at the approaches that have been taken to do introduce AOSD into the .Net framework. Here we examine the points of introduction , design and success of these implementations. We conclude by looking at the CME which is a new platform for the creation of AOSD implementations.

Chapter 3

In this chapter we explore the requirements for this project. We present the AspectC# design as a template for the design of a source code weaver. Finally, we present the design for SourceWeave.Net through presentation of the issues the design would have to overcome in the to achieve the aims of this work.

Chapter 4

This chapter looks at the implementation of SourceWeave.Net: The different components that make up the weaver, how they are physically designed, and what technologies were used to implement them.

Chapter 5

In this chapter we present an evaluation of SourceWeave.Net against other approaches aimed at introducing AOSD concepts to the .Net framework. This chapter also illustrates the limitations of this approach.

Chapter 6

This chapter concludes the thesis. In this chapter we present the benefits and limitations of cross-language source code weaving and conclude that this approach is a strong migration path for the introduction of AOSD to the .Net framework. We then present the work that we believe is needed to fully implement this proof-of-concept implementation of cross-language source code weaving.

Chapter 2

State of the Art Review

2.1 Introduction

The overall aim of this work is to introduce AOSD paradigm to the .Net platform. In this chapter we examine the .Net platform what it is and how it is structured. We also examine the AOSD implementations on both the Java and .Net platforms.

2.2 .Net Framework

The .Net framework exists at the core of the .Net initiative. The .Net initiative encompasses a suite of tools and technologies that will update the Microsoft platform for the current technology and business climate. Additionally, the .Net initiative represents a holistic approach to system design, creation, evolution and administration. The .Net framework is the base through which all aspects of the .Net Initiative are enabled to work together seamlessly[29].

The .Net framework is an architecture that enables a multi-language programming platform and it is based on a series of ECMA standards[18]. These standards specify a common infrastructure on which different programming languages can be implemented[6].

2.2.1 Objectives of the .Net Framework

In the highly decentralised environment of the Internet, rapid application development (RAD) is crucial. The framework has therefore been leveraged to achieve the following goals.

- To provide a consistent object oriented programming environment.

- To provide a code execution environment that minimises software deployment and versioning conflicts.
- To provide a code execution environment that guarantees safe execution of code.
- To provide a code execution environment that eliminates performance problems associated with scripting or interpreted environments.
- To make the developer experience consistent across widely varying types of applications.
- To build communications mechanisms on industry standards to ensure that code based on the .Net Framework can integrate with other code.

2.2.2 .Net Framework Architecture

The layered component architecture of the .Net framework is illustrated in Figure 2.1, the main component of which is Common Language Runtime (CLR) .

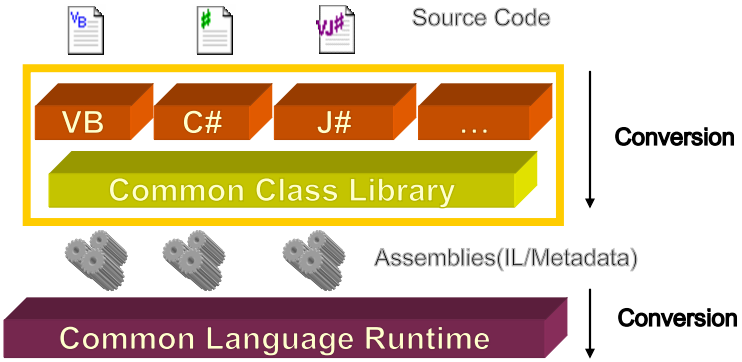

Figure 2.1: .Net framework architecture

2.2.2.1 Common Language Specification

The .Net Framework currently supports over 20 languages C#, VB.NET, C++, J#, Eiffel.Net, NetCobol, Small Talk, Perl, Python, Pascal, Oberon, Haskall, Mercury, Scheme, CAML, Component Pascal, Mondrian, Fortran, Dyalog ALP, CLisp ,OZ and F# in its common language model. All of these languages conform to the Common language specification(CLS).

The CLS defines conventions that languages must support in order to be inter-operable within .Net. By conformance to the CLS all, .Net compliant languages can compile to an common IL. Each language implementor provides a compiler that takes in source code and which outputs IL packaged as assemblies or modules. Figure 2.1 shows the compilers and CLL grouped inside a border. This border signifies the range of components the CLS specifies. The Common Type System (CTS) is a subset of the CLS.

Common Type System

The CTS defines standard, object oriented types and value types that are supported by all .Net compliant languages. It is the CTS that provides the unified programming model. The unified programming model denotes the ability of all .Net compliant languages to utilise common types and values as well as pass common types and values between inter-operating components written in different languages. A CTS is the most critical prerequisite for language inter-operation.

Metadata

Metadata is information created upon compilation of source code to an assembly which is stored as part of the assembly. Metadata information describes the assembly and the contents as well as other information such as the dependencies that the assembly has upon other assemblies. Metadata allows an assembly to be self-describing which allows the CLR to utilise such information to load classes, manage memory, debug, browse objects and translate IL to native code.

2.2.2.2 Common Class Library

The CCL is a set of base namespaces or libraries that can be utilised and extended by developers to create applications. These class namespaces are common to all .Net compliant languages. A developer can then utilise the same libraries when writing applications in different languages. This means that a developer only has to learn one set of APIs rather than having to know one per language.

2.2.2.3 Common Language Runtime

The CLR is the core of the .Net framework. The CLR acts as a virtual machine that manages code execution; it provides services such as memory management, thread management, and remoting. Also the CLR ensures strict type safety and enforces other forms of code accuracy that promote code security and robustness. Code that targets the CLR is known as managed code, while code that does not target the CLR is known as unmanaged code. The CLR takes assemblies that conform to the Common Language Infrastructure (CLI) and, using just-in-time compilers, converts the IL code into native code to run on a particular platform.

2.2.3 AOSD Integration.

To introduce AOSD to the .Net framework we must, extend some part of the framework or we must extend the framework itself. Each layer of the framework can be seen as a code translation or conversion point.

Figure 2.1 presents a layer of compilers as part of the .Net Framework Architecture. Each compiler takes source code written in a particular language that conforms to the CLS and converts it to a common IL. The CLR is another code translation point. The CLR takes the IL that conforms to the CLI, packaged as assemblies, and translates this to native code.

As we can see in Figure 2.2, to introduce AOSD by modifying the framework, we should either change the compilers or the CLR. Thus, the .Net languages could be extended to introduce new AOSD constructs. The compilers could then be modified to take the AOSD source code and compile it to OO IL. Furthermore the CLR could be changed to introduce AOSD concepts.

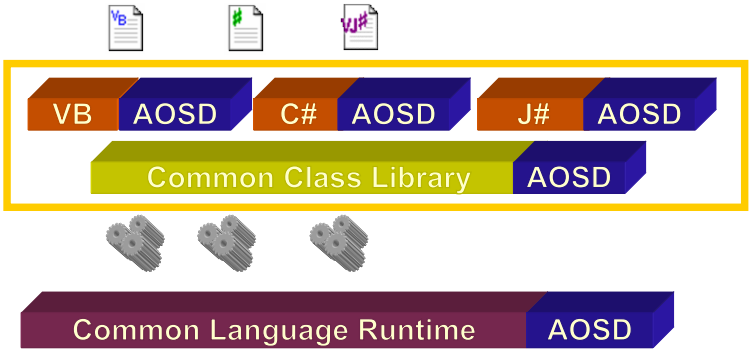

Figure 2.2: Introducing AODS through framework extension

The .Net Framework however is based upon common standards that are the basis of language interoperability. These standards have been developed around the OO paradigm and would need to be changed to support AOSD. Changing these standards for AOSD could be difficult as there may be resistance by parties such as language providers but also changing of the standards may force a rethink of the policies, such as security policies, specified within the standards. To introduce AOSD at a language level the CLS and CTS would have to be altered to take into account the new AOSD constructs or types. The CCL in this case may also have to be re-engineered with AOSD concepts in mind. To introduce AOSD at a run-time the CLI would have to be changed.

To introduce AOSD through extension of the framework we must add additional code translation points. One translation point can be introduced before compilation where source code with additional AOSD constructs could be translated to source code with OO constructs, which then can be compiled as normal.

Another translation point is post-compilation and pre-runt-time. Here assemblies representing distinct concerns can be composed resulting in woven assemblies. Then these assemblies can be loaded and executed by the CLR.

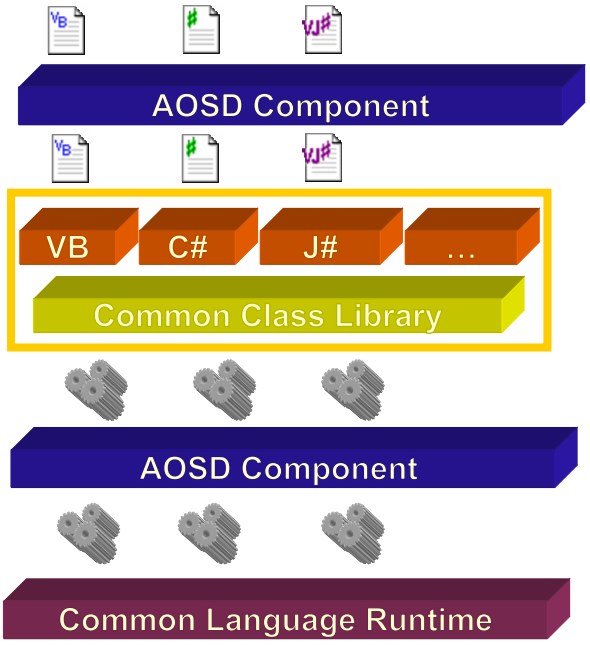

Figure 2.3: Introducing AODS through addition to framework

2.3 AOSD

Aspect Oriented Software Development (AOSD) is a relatively new paradigm that enhances the SOC, through modularisation of crosscutting concerns. AOSD has been implemented on many platforms using various modularisation and composition models. To position and quantify SourceWeave.Net we must firstly review the existing AOSD implementations.

2.3 Java Approaches

The migration of the Java platform to AOSD has begun with a variety of well-established different migration paths. The .Net approaches follow this work closely; most of the work in the field of AOSD has been achieved on the Java platform. Furthermore, the Java and .Net platforms are quite similar in architecture. These similarities with the positioning of Java at the forefront of the AOSD research community provide both a benchmark and a rich collection of migration paths that could be mirrored by AOSD approaches on the .Net platform.

2.3.1 AspectJ

Overview

AspectJ has emerged as the most popular migration path from OO to AOP for the Java platform.This year the AspectJ project moved from Xerox Parc[1] to Eclipse[5], and it has become an open source project, allowing the full involvement of the Java and aspect community. The popularity of this model is the result of the simplicity and intuitive features of AspectJ [7], and is partially due to the growing community support for AspectJ.

Language

Java is a single language platform. There has, however, been work devoted to mapping other languages to the Java Virtual machine [26]. The Java platform has taken the RISC approach and constrained its language base. Java being a single language platform has flexibility in terms of extending the Java language. Because Java is not constrained by standards and multi-party involvement, extension of the Java language is possible by altering the language and compiler. The compiler would the take source code with extended constructs map them to standard constructs in Java byte-code.

AspectJ has taken advantage of this single language freedom. It has extended the Java language with the introduction of new constructs to modularise crosscutting concerns. AspectJ has extended the Java compiler to introduce a weaving capability. This compiler then can weave the modularised crosscutting concerns into the base Java code and create standard woven OO Java byte-code.

Aspects

Aspects are modular units of crosscutting implementation[9]. This crosscutting implementation can be categorised into two distinct sets *:dynamic-crosscutting* and *static-crosscutting.*

Dynamic Crosscutting

Dynamic crosscutting in AspectJ is based on a joinpoint model. A *joinpoint* is a well-defined point of execution in the program. In a software system there are many *joinpoints.* However, there are only certain *joinpoints* wherein crosscutting behaviour should be introduced; these points are identified by *pointcuts.* A *pointcut* identifies a set of joinpoints in a program. Units of crosscutting behaviour are known as *advice.* Advice is behaviour that will be introduced to the system at joinpoints that are identified by pointcuts. Thus the dynamic crosscutting is the introduction of crosscutting behaviour to a system through the joinpoint model.

Joinpoint Model

AspectJ is recognised as having a very rich joinpoint model[20]. The joinpoint model is based on static code derived information. Some examples of AspectJ joinpoints are:

1. Method Call: is an execution point within code wherein a method with a particular signature is called.
2. Exception Handler: is an execution point within code where an exception of a particular type is caught.
3. Field Access: is an execution point within code at which a field is accessed.

A *pointcut* is used to identify these joinpoints as a pattern that can be matched against code, and a *pointcut designator* is the name given to that pattern. A number of primitive *designators* that can be composed by a number of operators to form complex user defined *designators*. AspectJ supports property-based pointcuts where pattern matching mechanisms such as wild-cards can be introduced to widen or narrow the joinpoint set the pointcut matches. This property-based matching allows a greater degree of control in the joinpoints a pointcut identifies. A pointcut also serves as a gateway between joinpoints and advice to define the context that needs to be exported from the base code to the aspect at run-time. A pointcut can be *named* or *anonymous*, a *named* pointcut exists independently as a construct, while an *anonymous* pointcut is declared within advice as it is anonymous and it cannot be referenced.

Advice is a method construct containing crosscutting behaviour. At run-time advice executes at sets of joinpoints defined by the pointcut that the advice references via name or direct anonymous reference. The advice executes before, around or after the actual behaviour represented by a joinpoint.

Both pointcuts and related advice are declared within an aspect. An aspect is a unit of modularisation that can contain similar declarations as classes, as well as AOP constructs.

Static Crosscutting

Static crosscutting is the *introduction* or transposition of characteristics (members, variables, interface declaration) that exist within an aspect into the base class. This transposition does not change existing behaviour; instead it has the power to add behaviour or change the relationship structure of the software system.

Critique

The AspectJ approach is now the most popular approach which enhances the ability to separate of concerns on the Java platform. AspectJ has been the focus of much attention from the Java community. This approach serves as the benchmark for AOSD implementations due to its success.

Given the simplicity, intuitiveness and richness the AspectJ joinpoint model it has become the basis for the implementation of many other AOSD implementations. SourceWeave.Net emulates the AspectJ joinpoint model because the advantages this model has.

2.3.2 Hyper/J

Overview

Hyper/J is a tool that has been developed to implement *hyperspaces* on the Java platform. Hyperspaces are the solution to the problem that is referred to as the "tyranny of dominant decomposition"[19] of classes in OO or functions in procedural languages. The Hyper/J tool works with standard Java bytecode. The current development does not implement the full hyperspaces standard, which therefore limits the full acceptance of Hyper/J by the Java community. The joinpoint model that is used in the Hyper/J model has been described as too simple [20].

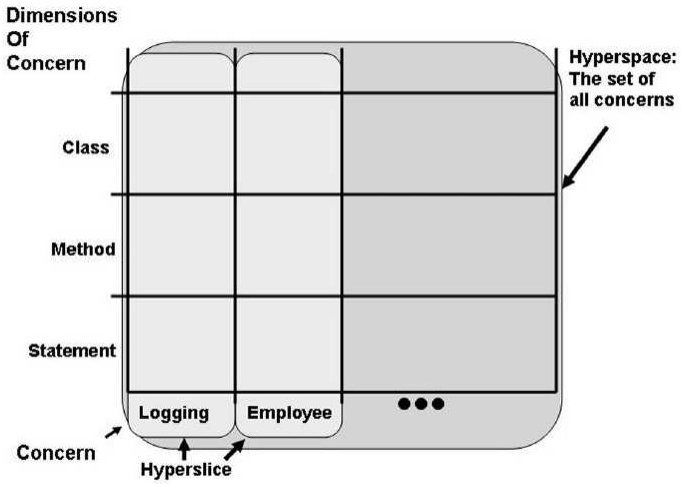

Figure 2.4: Hyper/J overview [Source: Hyper/J]

Hyperspaces

The goal of *hyperspaces* is to allow the multi-dimensional SOC [19]. Hyperspaces permit an explicit representation of all the important concerns in a system, and the identification and management of interactions among such concerns. The hyperspaces model aims to allow both separation and identification of concerns, thereby allowing the evolution of a system to which the concerns compose.

Concern Space

A *concern space* is a conceptual set of concerns that form in some body of software. These concerns may be primitive in nature, or a composition of primitive concerns to form a compound concern. The job of a concern space with respect to hyperspaces is to organise concerns in a fashion where all concerns are separated, and then to describe the relationship between concerns[4].

Hyperspace

A *hyperspace* is a concern space characterised by a number of key features in a multi-dimensional matrix where each axis represents a feature dimension and each point on that axis represents a concern. This matrix then exposes all concerns that exist on a particular dimension. Thus each dimension can be seen as a conceptual composition of the set of units. A hyperspace in Hyper/J maps the concerns it represents to an OO composition in the form of a class.

Hyperslice

A *hyperslice* is a concern that is declaratively complete[1], that is it must declare everything to which it refers. A hyperslice is required to exist in a distinguished dimension called a hyperslice dimension.

Hypermodule

A hypermodule is used to integrate a set of hyper slices. This integration is the binding of hyperslices, or the composition of hyperslices into a meaningful software artifact.

Critique

The hyperspaces model provides multi-dimensional SOC. However, due its early stage of implementation and lack of community involvement in comparison to AspectJ, it is difficult to quantify the value of this approach as it has not been fully testable.

The multi-dimensional SOC that Hyper/J proposes is now being looked at by the AspectJ community as an extension to the AspectJ model. Hyper/J has currently been dropped as a development project by its creator IBM. The AspectJ model has allowed Java developers separate concerns through provision of another modularisation dimension in the form of an Aspect. Extending this model to provide more separation dimensions may be the next step in the implementation of the multi-dimensional SOC.

[1] Declaratively complete means that anything that the concern references must be declared or encapsulated within the concern

2.3.3 ComposeJ

Overview

ComposeJ is an implementation of composition filters on the Java platform[28]. Currently ComposeJ is a command line tool that acts as an add-on to the Java compiler and contains a Java composition filter editor.

Composition Filters

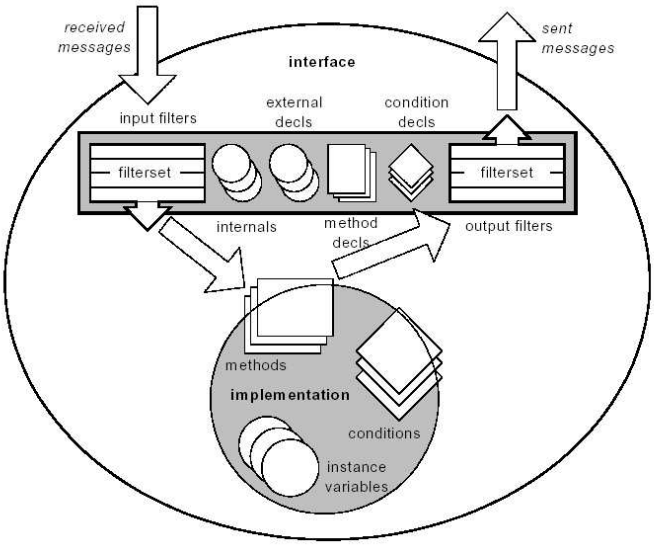

Figure 2.5: Composition filters [Source: ComposeJ]

Composition filters [2] is an approach to the composition of multiple concerns overlaying an OO model through object decoration. Composition filters act as transparent proxies that warp or decorate objects and intercept the calls in and out the objects which they decorate. A *filter* in this model is a modular unit of crosscutting implementation. A *filter* that intercepts a call to an object is called an *in* filter and a filter that intercepts a call from an object is called an *out* filter. The *filter* encapsulates a set of conditions that are related to units of behaviour. Once intercepted, the call is matched against these conditions. If the condition matches the call, the behaviour associated with that condition is executed. Once the conditions to be matched are exhausted, the call is propagated to either the target object, or to the next filter in the *filterset*. A *filterset* is a collection of filters that decorate an object.

Architecture

ComposeJ is implemented as a preprocessor to the Java compiler[28]. ComposeJ weaves proxy filters to decorate base classes. When a call to a particular class is invoked the decorator receives the call.

Crosscutting behaviour can be executed within the proxy, and the call may then be forwarded to the target object. ComposeJ only implements *in* filters as it cannot intercept calls fired from an object. The filter is not originally expressed in pure; Java it is expressed in a declarative language extension of Java and then converted into a Java class through the ComposeJ tool. In ComposeJ two types of filter error and dispatcher filters.

- Dispatch: if the call is accepted, it is dispatched to the current target of the message, otherwise the call continues to the subsequent filter.
- Error: if the filter rejects the call, it raises an exception, otherwise the call continues to the subsequent filter.

Critique

ComposeJ has been implemented at a source level as a preprocessor where it seems the model is possibly more suited to a run-time based implementation since ComposeJ cannot support *out* filters. By implementing Composition Filters at run-time *in* and *out* filters could be more easily supported.

Source level weaving is demonstrated here in a model that is possibly more suitable as a run-time implementation. This shows that source level weaving is both powerful and flexible. It also implies that implementing a weaver at the lower levels of a platform may be very difficult.

2.3.4 Discussion

In this section a subset of existing Java approaches are examined. All of these tools work at a source or byte-code level which, extends the Java language in some way.

Introduction of AOSD through modification of the JVM is difficult. Java is an open source platform, yet Sun have been protective about releasing the source code for their JVM out of the fear of a proliferation of JVM clones. Allowing the alteration of the JVM may impact the Java security model for example. Furthermore, the alteration of this low level code would be difficult. The attraction of Java has always been "write once run anywhere", altering a JVM would remove this proposition as the JVM alteration would have to be created for every OS.

Given that changing the JVM is a difficult, there are a few remaining options for AOSD implementation: a tool that weaves Java byte-code, a re-implementation of the Java compiler, or a source level weaver, or a combination of these. There are certain restrictions when using source code weaving in Java, in that it is only possible to weave Java source code, not weave byte-code. If there is no source for a particular component, it is impossible to weave new behaviour, or change the structure of that component. This raises the issue of code access security. A third party component supplier may not want their code to be crosscut with new behaviour. If the tool works at a byte-code level we are able to weave into

components regardless of source code access. Given the strength of the open source community on the Java platform, code access security is not as stringent a concern as in .Net. Both AspectJ and Hyper/J tools have developed language extensions and compilers that can compile down to standard Java byte-code. However, both approaches can also weave into byte-code; this is not the case with ComposeJ which is source dependent.

Therefore, source code weaving is the most popular weaving level for Java implementations of AOP tools. This is either done at a preprocessor to standard Java source code which targets a standard Java compiler, or it is done by creating a new compiler. This extended compiler would take modified Java grammar, which introduces new constructs into the Java language to modularise crosscutting concerns, and provide a way to create rules for re-composition of base and crosscutting concerns.

Modifying languages and compilers is an easier option for implementing AOSD in Java than modifying run time environments. All that this requires is a mapping from the modified language to standard byte-code.

Here there is also a possibility of cross-language weaving on the Java platform, by compiling a non-Java language to Java byte-code and then using some byte-code based Java AOSD tool weave in cross cutting concerns expressed in Java.

2.4 Related Work

Many approaches to implementing AOSD exist and are currently emerging. This section examines these various approaches. This will serve as an insight to the concepts and motivation behind this work. In Figure 2.6, we can see the positioning of the approaches examined here within the .Net framework architecture.

2.4.1 AspectC#

Overview

AspectC# is a AOSD extension to the .Net framework developed by Howard Kim and it is the basis for this work. AspectC# enables a developer to modularise crosscutting concerns and compose those concerns with base concerns within the C# language.

AspectC# is a source level weaver for the C# language that is loosely based on the AspectJ joinpoint model. As we can see in figure 2.6, AspectC# is implemented as a preprocessor or weaver that creates a mapping of AOP concepts superimposed on top of the C# compiler. The aim of creating this tool as a preprocessor to the C# compiler has been to avoid modifying the C# language. The underlying reasoning is that AOP should augment and extend the modularisation available, but not change the OO model which is the basis of the C# language.

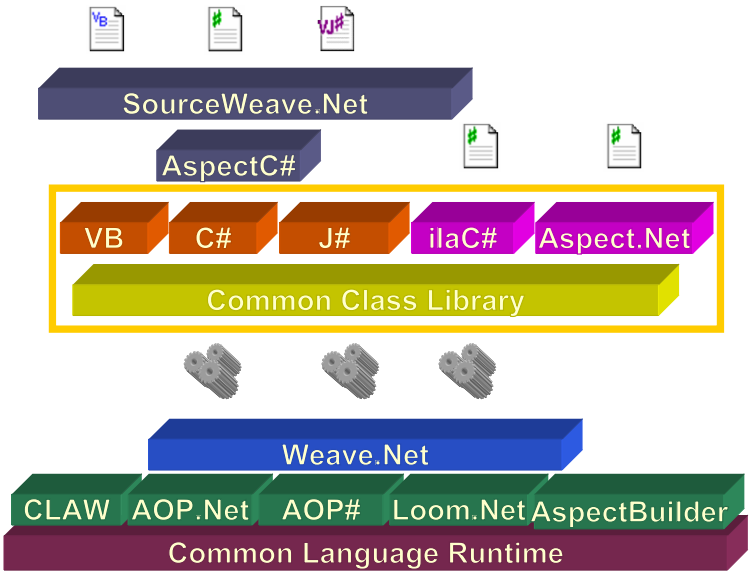

Figure 2.6: .Net AOSD Approaches

The .Net framework has been designed around the OO paradigm. An alteration to a single compiler, in this case a the C# compiler, would cause a major change in the CLS if language inter-operability was to be retained. By implementing AOP in C# as a preprocessor , there is no alteration to the .Net framework. Through this approach, the stability of the CLS is maintained and the .Net framework remains completely unaffected. From this perspective, the AspectC# positioning above the .Net framework is a sensible option [11].

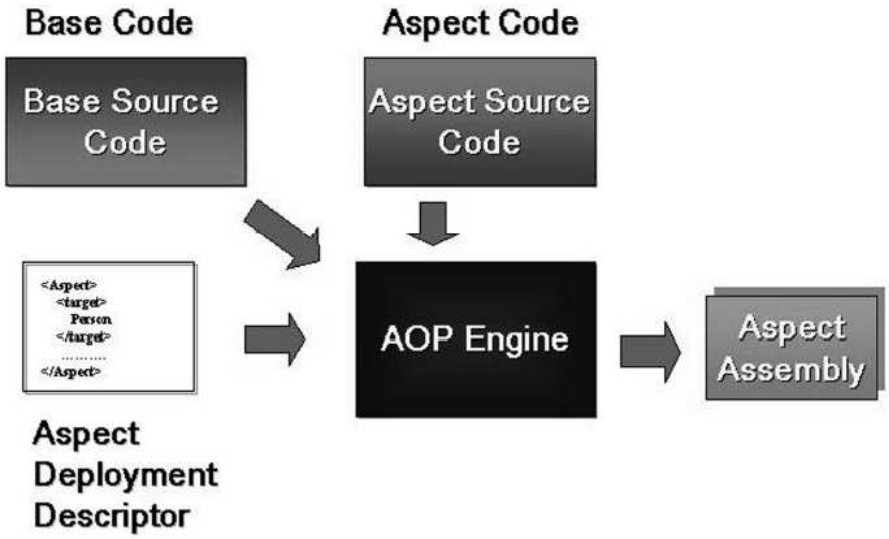

Figure 2.7: AspectCSharp Project Architecture [Source: AspectCSharp]

Joinpoint Model

AspectC# has loosely similar semantics to AspectJ. It supports two types of crosscutting concerns, both dynamic and static. AspectC# encapsulates a concern in an aspect class. The methods in the aspect class represent advice or crosscutting behaviour. The actual aspect characteristics are not encapsulated with the crosscutting behaviour, instead they are expressed in a separate XML aspect descriptor. This aspect descriptor maps the crosscutting behaviour encapsulated within the advice methods of the aspect class to joinpoints found within the base classes. AspectC# has implemented one joinpoint type, an *execution* joinpoint.

Architecture

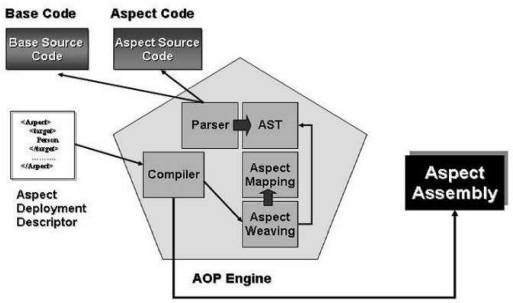

Figure 2.8: AspectCSharp Architecture [Source: AspectCSharp]

The AspectC# architecture presented in Figure 2.8 is based on the concept of an Abstract Syntax Tree (AST), provided through the CodeDOM API. This AST is created by parsing base code and source code to the CodeDOM AST. The Aspect Mapping is responsible for interpreting the Aspect XML Descriptor to discover the mapping from the aspect code to the base code. The aspect weaver then weaves or merges the AST's of both the base code AST and the aspect code AST. The Compiler then takes the merged AST and creates an assembly.

Critique

The AspectC# tool has some weaknesses in its implementation. The major weakness is the naive weaving implementation where the declarative completeness is lost during weaving. Moreover, the joinpoint model is limited and not naturally extensible.

The AspectC# XML Aspect Descriptor is constrictive by design. It references file locations directly which restricts portability. It supports only the expression of an execution pointcut, and both joinpoint composition and property based joinpoint identification are not supported in this model.

The AspectC# weaving model was naive, in that the AST created from the Aspect class in terms of the code contained in the advice methods was merely copied from the advice methods directly into the

base methods. The significance of this is that autonomy and encapsulation of the Aspect is broken, and as such declarative completeness is lost post-weave. If we are treating an aspect as an autonomous module of coupled attributes and functionality, then by merely copying the internals of the modules advice methods, this coupling is lost.

AspectC# is the foundation of this work, since it provides a basis upon which cross-language weaving at source level can be achieved.

2.4.2 ilaC#

Overview

Instance level aspects for C# [20] is a project that aims to combine expressive pointcut languages and instance level weaving. Essentially this project aims to allow the developer say "weave this aspect to only these instances of this class".

Architecture

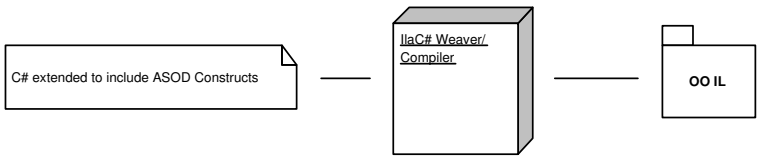

Figure 2.9: IlaCSharp

IlaC# aims to create and extend the C# language through introducing keywords akin to those in the AspectJ language extension. The purpose of this is to achieve an expressive joinpoint model in a manner similar to AspectJ. IlaC# introduces a new keyword, *instancelevel* which is not part of the AspectJ language. The keyword will separate dynamic weaving from instance based run-time weaving. Theoretically this run-time instance level weaving would be implemented through joinpoint decoration at compile-time. The prototype weaver uses the CodeDOM namespace in the implementation of their new language.

IlacC# is working at the compiler level in the creation of the new language. Hence, the IlaC# weaver works with source code, the AOSD constructs which allow the modularisation of crosscutting concerns. These crosscutting concerns are composed with base concerns at the relevant joinpoints. IlaC# implements the AspectJ model for the C# language, however with the instance based weaving a variation of the composition filters model is being used.

This approach is a single language model, which is not concerned with cross-language weaving.

Critique

IlaC# is similar to the approach that was firstly introduced by AspectC# and continued by SourceWeave.Net, in the use of CodeDOM to support source code weaving. The focus of IlaC# however, is very different as cross-language weaving is not a concern. Finally, IlaC# shows that there is confidence that source code weaving can support a rich and extensible joinpoint model.

2.4.3 Aspect.Net

Overview

Aspect.Net [21] is a project that intends to analyse and extend the C# language to incorporate AOSD constructs into C#. Included as part of this research is an investigation into the CLI and CLR to investigate how the .Net framework from its core out could incorporate AOSD.

Architecture

The project hopes to create a tool for weaving the C# language with AOP extensions. Moreover it aims to create a new set of IL metadata that could be used at the CLR layer. This work is being done on Rotor Shared Source CLI [15].

Critique Aspect.Net aims to extend the C# language to introduce AOSD constructs. This is very similar to the approach that is being taken by IlaC#, however no weaver has been released as of yet.

2.4.4 Weave.Net

Overview

Weave.Net[13] is a load-time IL weaver that is currently being implemented by Donal Lafferty. The aim of this project is to achieve language-independent AOP with an extensive joinpoint model. The tool then relies on the CLI as the inputs to the weaver are assemblies.

Architecture

A precursor to this project was the development of an W3C XML schema language specification[12] for aspect specific constructs. This XML schema, derived from AspectJ syntax serves to describe the crosscutting characteristics of an aspect.

An aspect and a class are very similar types since both encapsulate members. However, an aspect has members which contain crosscutting behaviour and structure. The difference between an aspect and a

class is the additional syntax that allows the selection of joinpoints where the crosscutting behaviour is introduced.

This W3C XML schema allows the separation of these characteristics out of an aspect type as an XML representation. This representation can then be used to describe the crosscutting behaviour that can be encapsulated within a class. This is achieved through a reference within the XML to the class which contains crosscutting behaviour. In effect, there is a separation between behaviour and aspect characterisation.

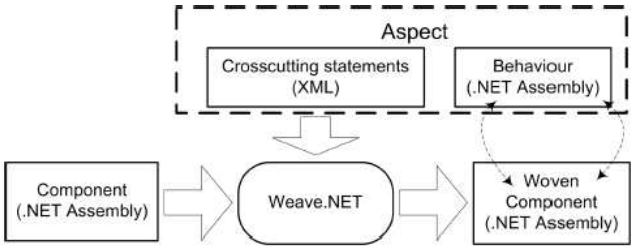

Figure 2.10: Weave.Net [Source: Weave.Net]

The architecture of Weave.Net is based on the XML representation of the AspectJ constructs. The aforementioned project is a load-time weaver, which means that the weaver is used post-compilation and pre-run-time. The weaver uses reflection to discover joinpoints from exposed metadata and an IL analysis. This analysis is derived from the XML declaration which describes how joinpoints are to be identified. Once joinpoints are identified a new assembly is created through utilisation of the Reflection.Emit API. The new assembly is a composition of the original base IL crosscut at joinpoints with references to the modularised crosscutting behaviour.

Critique

Weave.Net implements a rich joinpoint model based on the AspectJ joinpoint model. There is a constraint, however, since Weave.Net can only work with CLI compliant assemblies. Weave.Net cannot work with languages such as J#, as the output of the J# compiler does not strictly conform to the CLI. Weave.Net modifies IL and thus the mapping a compiler makes between source code and IL on compilation to enable debugging is lost through weaving IL. This mapping source code debugging. Since the source code-IL mapping is disturbed woven assemblies cannot be debugged. Weave.Net and SourceWeave.Net share a common joinpoint model through usage of the same aspect specification. However, some of the constructs such as the proceed construct related with *around* advice cannot be implemented in Weave.Net but are supported by SourceWeave.Net.

2.4.5 CLAW/RAW

Overview

The Cross-language Aspect Weaver (CLAW) [14] is a prototype run-time AOP based approach. The Run-time Aspect Weaver (RAW) tool implementation has been abandoned due to implementation difficulty.

Architecture

The RAW tool loosely shares a similar but less expressive pointcut model with AspectJ. In RAW, weaving process is carried out at run-time. The tool weaves assemblies that conform to the CLI. RAW uses an XML description to characterise aspects crosscutting characteristics and map the behaviour to an assemblies behaviour.

Raw weaves code prior to the Just-In-Time (JIT) compilation. RAW achieves this by providing a weave instructions file, which is compiled from the XML aspect description, to the RAW tool to identify points at which crosscutting behaviour should be composed into the base application. Before the compilation of a method, the signature is checked against the weave instruction set. If the method is to be crosscut with aspect behaviour, the method's IL code is converted to an AST, and thereafter the crosscutting behaviour is woven into the AST, and the resulting IL is emitted to memory for JIT compilation.

RAW uses the unmanaged profiling API to hook into the CLR through the ICorProfilerCallback and ICorProfilerCallbackInfo interfaces. Profilers need to remain unmanaged to allow the weaver perform actions unavailable or inadvisable within a managed environment. Profiling utilises an event-based system, through the described interfaces, the raw tool allows the handling of relevant events. These events provide the context information needed for weaving at run-time.

The meta information retrieved at run-time is very primitive and difficult to access. Much of the work in this project was to directed to creating a library (much like the reflection library of .Net) to allow easy access to this meta information.

Critique

The development of CLAW has been almost abandoned. The usage of unmanaged entry points to the .Net framework seems to be the major drawback in this work. This work shows that hooking into the CLR through unmanaged interfaces leaves the approach without much of the support that remaining within the managed environment that .Net provides.

2.4.6 AOP#

Overview

AOP# was a project which began as the diploma thesis of Mario Schupany. An implementation was begun but was never completed. The aim of the project was to create an AOP tool that would work without extending languages to introduce new AOP constructs. The aims of AOP# included a complete separation of base code and aspect code, a simple composition mapping mechanism and the introduction of aspectual polymorphism [2].

Architecture

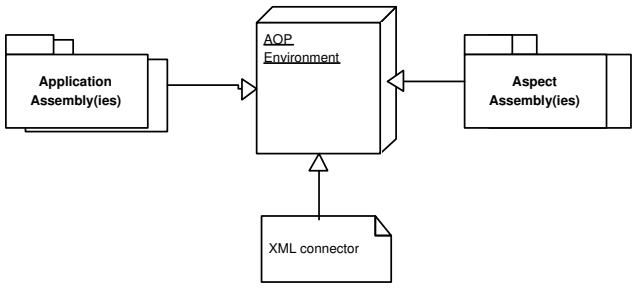

Figure 2.11: AOP-Sharp Architecture

AOP# works with base and aspect assemblies which contain the base and crosscutting concerns. These concerns are connected through declaration within an XML connector which describes the relationship between concerns. As illustrated in figure 2.8, the approach AOP# took was based around modularising base code and aspect code as OO IL and by using an XML *connector* as a composition mapping script. The AOP-environment would then take these artifacts at load-time and using the *connector* to weave the base and aspect code. AOP# bases its implementation on the utilisation of assembly metadata retrieved through the profiling API. This information id used to analyse IL and weave concerns.

Critique

The usage of the .Net profiling API restricts the AOP# idea to only working with managed code, as the .Net profiling API can only work with IL. According to Fabian Schmied [22], this work was not thought through and has been abandoned. Source weaving is in theory not constrained to working with only managed code, however in practice this is very hard to implement as cross-language source code weaving is possible due to standardisation. Unmanaged code does not conform to the .Net standards and thus is difficult to support.

[2] Aspectual polymorphism relates to the ability of turning on and off aspects at runtime.

2.4.7 AOP.Net

Overview

AOP.Net is a follow up to the AOP# project. Thus, the aims of AOP.Net are similar to AOP#.

Architecture

The AOP.Net architecture is based on the unmanaged profiling API, and as such, it is implemented as a COM component, which is notified when run-time events occur. Because of this architecture, AOP.Net cannot take advantage of the .Net class library. This in turn leads to the utilisation of the primitive unmanaged metadata. The technologies that AOP.Net is built upon make it impossible to get direct access to source code for analysis and manipulation. Thus, only after and before advice can be composed at signature joinpoints in base code.

Critique

The main problem with this approach is the lack of support for analysis and manipulation of .Net IL code. Being outside the .Net framework restricts the concern composition model. Because of this restriction AOP.Net is constrained in terms of the joinpoint model that it can support. Not being able to analyse construct to a base level means that a range of joinpoints cannot supported. Concerns that crosscut execution points that are not identifiable cannot be modularised. Thus, there is less opportunity to separate concerns.

2.4.8 Loom.Net

Overview

Loom.Net is a tool that has evolved from another AOP tool called Wrapper Assistant [23]. Loom.Net [27] is an AOP tool that is reminiscent of the composition filter model. In this approach, crosscutting concerns are modularised as proxies or filters. At run-time these proxies intercept calls on objects representing base concerns and the proxy object executes the crosscutting behaviour which it encapsulates. Loom.Net supports a limited set of crosscutting concerns as aspect specific templates. An aspect specific template is a standardised type of concern. Aspect specific templates provide a way to write proxies using macros that map declarative elements within the components being crosscut.

Architecture

This tool works at an IL level, and rather than working with the Reflection namespace available, the weaving is based on metadata attained through COM profiler interfaces. The tool generates proxy code

that maps the interface of base code; however, the code is not regenerated within the proxy. The proxy merely contains the composition of the base code and the code that is modularised or separated as a concern. This tool takes a base assembly and weaves an aspect assembly; the result is a assembly in which the OO constructs are proxied with crosscutting behaviour.

The mapping of base and crosscutting concerns is achieved through joinpoint declaration within the base code. This declaration is in the form of CLI custom attributes.

Critique

This tool is restrictive in that only certain crosscutting concerns can be modularised in aspect specific templates. Making joinpoints explicit through declaration is also restrictive as in a large system this may be infeasible.

2.4.9 Aspect Builder

Overview

Aspect Builder [3] is a tool that also follows the composition filter model. But unlike Loom.Net, described in section 2.4.8, weaving occurs at run-time. In this a filter is a represents a crosscutting concern and is a modularisation of crosscutting behaviour. In this approach a transparent filter intercepts calls and based on the call execute crosscutting behaviour.

Architecture

This architecture is based on an interception model. The AspectBuilder implementation is based around the use of the class ContextBoundObject as a foundation for a filter or a transparent proxy object. In this architecture calls on the stack are intercepted by a sub class of type ContexctBoundObject, the call is processed, and any actions that related to the call are carried out. Once the transparent proxy has finished processing its concern action based upon the call, it returns the call to the call stack. Custom assembly information associated with methods allows the association between base code and the filter or proxy through particular contexts.

Critique

This approach is intrusive as base code must have associated properties, such that contexts can associate the base code execution with particular filters. The need to modify base concerns to facilitate introduction of crosscutting behaviour is restrictive, thus this solution is not scalable.

2.4.10 Discussion

Implementation within the .Net Framework As we can see from the AspectBuilder implementation the implementation of AOSD as a run-time modification is possible. This is due the availability of interfaces that through extension can be leveraged for the implementation of transparent proxies.

There are many implementations such as CLAW, AOP.Net, AOP# and Loom.Net that add an additional code translation point to the .Net framework to introduce AOSD. These range in implementation, from utilisation of managed and unmanaged profilers to reflection with which to identify joinpoints and weave crosscutting behaviour. These approaches are however similar, as they achieve AOSD implementation through modification of IL. Because of the dependency these approaches have on IL, they are constrained by the CLI. These approaches cannot then work with languages that do not compile to strict CLI nor can they easily support debugging. Beyond this, due to the lack of support for code analysis, some approaches are unable to support rich joinpoint models. CLAW is a good example of this, in that most of the implementation spent on CLAW was in the creation of a library for code analysis. The development of CLAW was abandoned because of this. These approaches are language-independent within the bounds of the CLI in that all languages compile to the same IL. In weaving IL the language that the IL was created from is inconsequential.

Both Aspect.Net and IlaC# aim to introduce AOSD to C# through extension of the C# language to add AOSD constructs. This approach forces an extension of the C# compiler to implement weaving functionality. This weaving functionality provides a mapping from C# AOSD constructs to the standard CLI OO. This will allow language inter-operability but cross-language weaving would not be supported. Rich joinpoint models can be implemented in these approaches as the source code can be easily parsed to discover joinpoints and also source code is malleable and easily modified which is important for weaving.

AspectC# introduced AOSD concepts into the C# language without language modification. AspectC# is a code translation point at which cross cutting behaviour, modularised within OO C# constructs, is composed at execution points in base source code. This is achieved through declaratively expressing the crosscutting characteristics of the crosscutting behaviour and using this information to weave source code. This approach has all the advantages of source code weaving of language modification but language extension is avoided and .Net standards are maintained.

There are many approaches that add code translation points to work with .Net artifacts to implement AOSD in .Net. These artifacts are based on the OO paradigm. There are then no AOSD constructs which can be expressed within theses artifacts. Thus these AOSD constructs must be then expressed around OO constructs. Crosscutting behaviour must be encapsulated in some construct; many approaches have used classes to modularise this behaviour and then expressed the crosscutting characteristics of that behaviour in a separate XML file. AspectC#, Weave.Net, CLAW, AOP.Net and AOP# are all examples of this.

Approaches taken The concepts behind many of the AOP tools implemented for .Net have been mainly based on the AspectJ joinpoint model. Its simplicity to implement and to understand are a real draw to the AspectJ model. The most pure emulation of the AspectJ has been Weave.Net. Many approaches claim to follow the AspectJ joinpoint model, but in reality, they can only support a limited set of the joinpoints that AspectJ supports.

The composition filters approach has been emulated by Aspect Builder and Loom.Net. The composition filters approach does lend itself to a run-time implementation. This approach has been difficult in Java but looks much more of a viable option for .Net.

Surprisingly, there have been no attempts to implement hyperspaces in .Net. This is in part due the increased complexity of the hyperspaces model, the lack of support in terms of community information dispersal, and the lack of implementation of the hyperspaces specification.

2.5 Concern Manipulation Environment

2.5.1 Overview

IBM research has begun to look at AOSD in terms of the software development life-cycle. They have identified many of the AOSD approaches as having similar, if not common, implementation needs. Also identified are a set of base components that are common to many of the AOSD approaches that exist today. IBM has begun to create an environment wherein those who wish to pursue new approaches or standardise existing approaches could utilise their Concern Manipulation Environment(CME) to create new variations of the current AOSD implementations. The CME has been proposed to create tools which support the entire software life-cycle using AOSD concepts. [25]

2.5.2 Language Independence

The CME is of interest as it is not only taking on AOSD concepts and examining how they are applied throughout the software life cycle, but also because this work aims at making the CME language-independent. As this language independence is one of the major objectives of this project, the CME is of interest in that not only does it intend to break the platform boundary, but it also seeks to support multiple AOSD paradigms and environments. The CME was released during the course of this work and was considered as a platform upon which to explore language-independent source code weaving. However, the CME lacked the maturity and support required to be the used in this work. [25]

2.5.3 Future Benefits

With the growing plethora of AOSD approaches and post object paradigms, this work is of value to the AOSD community. It will promote a degree of standardisation in a paradigm that is quickly maturing

and gathering momentum. Such standardisation will provide a platform where it may be a possibility of differing AOSD paradigms being used in conjunction to separate concerns in systems that are written in differing languages.

2.6 Summary

In this chapter we looked at the .Net framework, what it is and the .Net framework architecture. We then explored how this architecture could be altered or extended to introduce the AOSD into the framework. We then examined how differing models of AOSD have been introduced on the Java platform. We then look at the approaches that have been taken to introduce AOSD into the .Net framework. We examined the points of introduction , design and success of these implementations. We concluded by looking at the CME which is a new platform for the creation of AOSD implementations.

Chapter 3

Design

3.1 Introduction

SourceWeave.Net aimed to implement a cross-language source code weaver that supported a rich joinpoint model. In this chapter we look at the design we have created to achieve this aim.

3.2 Requirements

Here we present the requirements that SourceWeave.Net had to implement to fulfil the aims of this work.

1. SourceWeave.Net must allow the developer to modularise crosscutting concerns within .Net compliant languages.
2. SourceWeave.Net must allow for the composition of base and crosscutting concerns.
3. The tool should make no language extensions to any .Net compliant languages.
4. This tool should be extensible beyond its current scope for future research based projects.
5. SourceWeave.Net must implement an extensive joinpoint model.
6. Inputs to the system must not be altered.
7. This tool should be easy and intuitive to use and should be in keeping with the .Net base objectives of simplicity and rapid development.
8. An evaluation of this project as a migration path for .Net from OO to AOP is a requirement
9. The tool must run on the .Net framework.

3.3 AspectC#

3.3.1 SourceWeave.Net and AspectC#

The design of SourceWeave.Net has followed the general approach of AspectC#. AspectC#'s model was designed to enable the weaving of modularised concerns across base concerns within source code written in the C# language. SourceWeave.Net uses the AspectC# design as a template which is extended to achieve cross-language weaving with a rich joinpoint model.

3.3.2 AspectC# Design

3.3.2.1 External View

AspectC# identified a number of components, which are required in the design of a source-code-base weaver for a single language scenario. AspectC# takes as input C# source and an XML aspect descriptor. The XML aspect descriptor identifies crosscutting and base behaviour within the source code, as well as specifying how to weave these together. The output after weaving would then be an assembly, which would contain the crosscutting behaviour distributed across the base code.

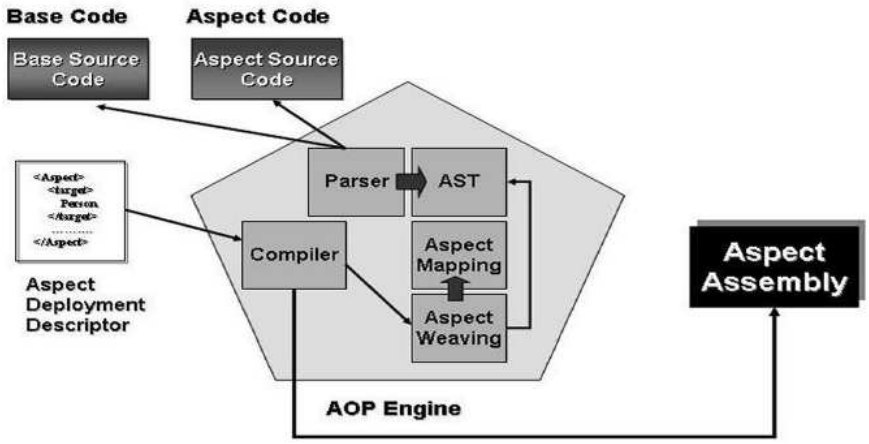

Figure 3.1: AspectCSharp component architecture [Source: AspectCSharp]

3.3.2.2 Components

Figure 3.1 illustrates the components in the AspectC# design. These components are described below.

Parser: The parser component converts the C# source code into an AST.

AST: The AST represents the source code abstractly.

Aspect Mapping: The Aspect Mapping component identifies the joinpoints within the AST and checks to see if they are to be crosscut.

Aspect Weaver: The Aspect Weaver component weaves cross cutting behaviour at joinpoints to be crosscut.

Compiler: The compiler compiles the woven source code to an assembly.

3.3.3 Design Evaluation

The components which make up the AspectC# design enable modularisation of crosscutting concerns and weaving of these concerns across base concerns within the C# language. This design follows the architecture of other AOSD implementations such as CLAW and AOP# discussed in Chapter 2. Like these approaches, AspectC# utilises the OO class construct to modularise crosscutting behaviour and then separately represents the characteristics of this crosscutting behaviour declaratively in XML. However this design was C# and source code based, where as the other tools that had taken this approach were IL based and language-independent. This XML extension of the OO class construct is the basis for the joinpoint model in this type of approach.

3.3.3.1 Joinpoint Model

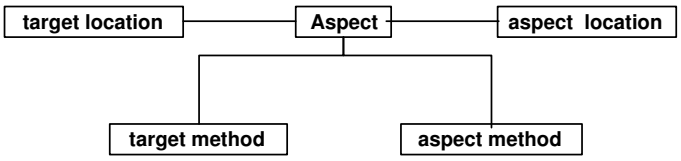

Figure 3.2: AspectCSharp Joinpoint Model

AspectC# did not aim to implement an extensive joinpoint model. The AspectC# XML aspect descriptor only supported one type of pointcut and consequently the joinpoint model could only support this joinpoint. This limitation then also negated the possibility of supporting context sharing between aspect and base behaviour as well as joinpoint composition.

3.3.3.2 Weaving Model

AspectC#[11] claimed that aspects were declarative complete within the AspectC# model. From a modularisation perspective this was true, but as we can see in Figure 3.3, declarative completeness is broken post-weave. AspectC#'s weaving model is implemented through copying the crosscutting code

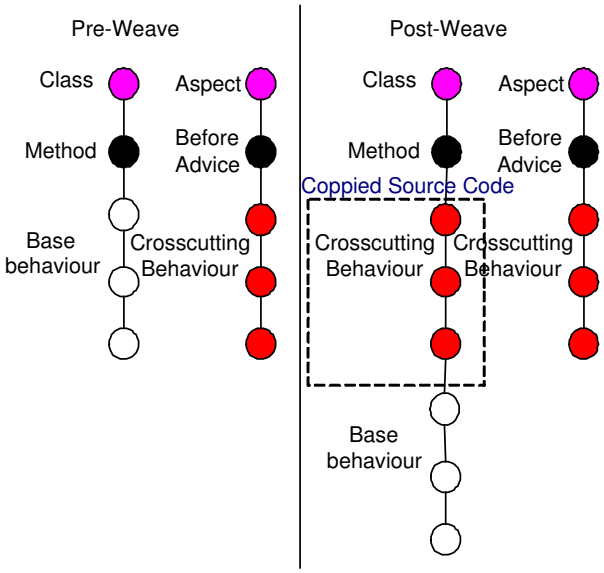

Figure 3.3: AspectCSharp Weaving Model

from the aspect to the base code. This is unsafe, as there may be references within that code to other members (fields,methods) of the aspect, or there may be references in locals to types not declared within the scope of the type containing the joinpoint. Therefore, if a developer using this model must ensure that the advice the write must be completely isolated and the identifier naming used does not conflict with any of the identifier names at the joinpoints where that code is to be woven.

3.3.3.3 Conclusion

Extension of this model would not support cross-language weaving. AspectC# assumed that cross-language weaving could be implemented through creation of more parsers to CodeDOM. This however is not the case. There are major issues in relation to parsing, joinpoint identification, weaving and compilation that this design could not deal with. The major difference is the change in the nature of the inputs and outputs in a cross-language weaving scenario, as will be described in the next section.

3.4 SourceWeave.Net Design

SourceWeave.Net loosely follows AspectC#'s design to implement source code weaving. The AspectC# design identifies common components that are needed within the design of a source code weaver.

- A parser to either find joinpoints or to convert the source code into a format where joinpoints can be discovered.
- An AST component is needed in any situation where cross-language weaving needs to be applied.

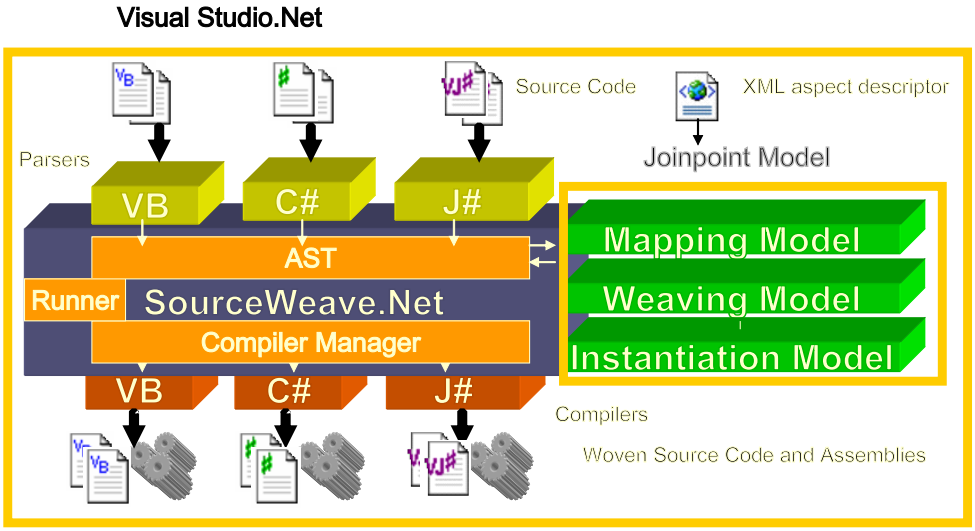

Figure 3.4: SourceWeave.Net Logical Design

- A mapping model and weaving model which are composed to form a joinpoint model which allows concern modularisation and weaving.
- A compiler is needed to compile woven source code.

However, SourceWeave.Net is required to implement cross-language weaving and support a rich joinpoint model. The design of SourceWeave.Net then had to cater for these requirements. In addition, we wanted to match our design to the objectives that the .Net framework was designed to achieve. For this reason SourceWeave.Net was designed with RAD in mind.

3.4.1 Introduction - Source Code Weaving

Figure 3.4 presents the overall design of SourceWeave.Net. We can see that the design is enclosed in a border, this is to signify that SourceWeave.Net has been designed as a plugin to Visual Studio.Net (VS). There are many reasons for this, the most important of which is the standardised structuring of source code and assemblies that the VS environment provides. This structure allows automatic detection of input and output locations. As SourceWeave.Net is working with source code, being able to locate source files automatically that hold is advantageous for a developer, as there is no need to give explicit file references as input to the weaver. This automation is also in-line with the .Net objectives of simplicity and RAD.

As was discussed in Section 2.2.2, a programmer can write source code in many languages that conform to the CLS and compile the source code with a language specific compiler to IL. This has an affect on how source code is organised. C# source code, for example can only be compiled by a C# compiler which means that any component that is made up of a number of source code files need to be organised

together as a unit for compilation. VS provides a structured way to achieve this source file organisation. When writing an application in VS a developer must create a solution. A solution is a root structure in which projects can be contained. A project can be seen as a unit of source code to be compiled by a specific compiler. Thus a developer can create a VB project or a C# project within a solution. Within a C# project one can create or add C# source files, for example. Each project also has a list of references to other projects or assemblies. That is, if the source code references types expressed within source code packaged within projects or assemblies, a reference must be made to those projects or assemblies. This is to enable the compilation unit in which the source code exists in to be compiled. This structuring is mirrored on the file system as a directory structure. This standardisation allows SourceWeave.Net to be designed to automatically discover source code and discover the language the source code is written in.

3.4.1.1 Cross-Language Weaving - Parsers

Because SourceWeave.Net aims to implement cross-language source code weaving, the source code needs to be abstracted to the point where the source code is represented in a common way. In the design this is achieved by having a set of parsers, one for each language, where the parser converts the source code written in a particular language to an AST. The AST is then a language-independent representation of the source code. This creation of the AST is then the basis for cross-language weaving as the joinpoint model can treat all the source code in a uniform way and thus weave behaviour originally expressed in one language into behaviour expressed in a completely different language.

3.4.1.2 Joinpoint Model

SourceWeave.Net aimed to support a rich joinpoint model. The SourceWeave.Net joinpoint model is presented in Figure 3.4, we can see that the joinpoint model is composed of three models a mapping model, a weaving model and an instantiation model. Also we see that the joinpoint model works with the AST's which represents the source code in a language-independent way.

The joinpoint model is based on an XML specification [12] that allows the expression of the crosscutting characteristics of an aspect. This specification was developed by Donal Lafferty and is used in the design and implementation of Weave.Net. The XML specification is a language neutral representation of the AspectJ syntax. Through conformance to this specification we achieve the separation of crosscutting behaviour within a OO class construct and the characteristics of that crosscutting behaviour in an XML document. Thus we can use the OO constructs that the .Net framework languages are based on and use XML to declaratively specify how the input source code should be woven. In this way there is no extension to any language when introducing AOSD concepts to the .Net programming platform.

This specification represents the full set of joinpoints, advice types and introduction types that are available in the AspectJ model. As such this specification is the basis for an extensive joinpoint model.

Given that we can modularise crosscutting behaviour within a class and separately declare the crosscutting characteristics of that behaviour, the design of SourceWeave.Net must now provide models for the composition of the modularised crosscutting concern with base concerns that are also modularised within class constructs.

Mapping Model

To weave this crosscutting behaviour across the base concerns, we must firstly be able to identify the joinpoints within the abstracted source code (ASC) that are to be crosscut with this behaviour. To cater for this, a mapping model is included as part of the design. The mapping model uses the XML specification which supports the declaration of pointcuts within the XML characterisation of the crosscutting concern to identify the joinpoints within base ASC to be crosscut. Both primitive and composite pointcuts are supported within this specification. The primitive pointcuts can also be property based under this specification, that is the pointcut can declared with special operators that allow selection of points in source code based on properties of the ASC. Also under this specification primitive pointcuts can be composed to create more elaborate pointcuts. Composite pointcuts allow the introduction of more criteria for joinpoint identification. With this matching model the ASC syntax represented by a joinpoint can be matched against the pointcuts, if the source code fits the pattern specified within a pointcut then this is a joinpoint at which crosscutting behaviour is to be woven. Because of the range of primitive pointcut types supported and the support for composition and property based pointcut declaration this specification allows is the foundation for the identification of both broad and focused joinpoints within the ASC. This in turn equates to a greater opportunity to separate and modularise crosscutting concerns.

Weaving Model

For joinpoints that have been matched there must be a way to weave the crosscutting behaviour into the source code which the joinpoint represents. The weaving model is part of the design which governs this action. The weaving model uses the XML specification which supports declaration type and location of advice. Thus the weaving model uses this information to weave the crosscutting source code into the base source code at the joinpoint. The specification also supports the passing of context from the base behaviour to the crosscutting behaviour, this is supported through the instantiation model.

The weaving model also supports weaving at a higher level to sustain cross-language source code weaving. Because the compile units which are input into SourceWeave.Net are converted to abstracted compile units (ACU) there not only must there be source weaving but also reference weaving between ACU's. Thus references to aspect ACU must be woven into base ACU that are crosscut by an aspect ACU. Figure 3.5, demonstrates what the weaving model should achieve. Pre-weave there is no association between the base and aspect ACU and base ACU but post weave these associations have been woven in

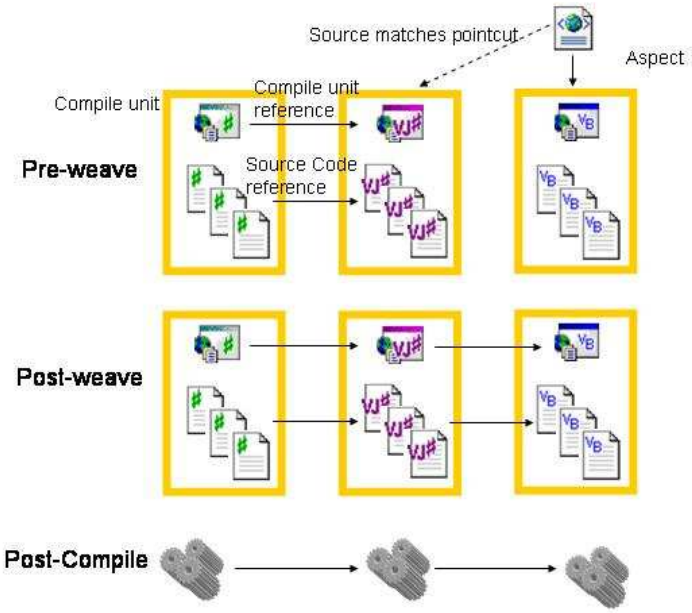

Figure 3.5: Reference Weaving

to the base ACU. These references between ACUs is reflected post-compile-time in the references that exist between assemblies.

Instantiation Model

The instantiation model has been included as part of the design of the joinpoint model firstly to support declarative completeness pre and post weave, but also to support the passing of context between base and crosscutting behaviour.

3.4.1.3 Cross-Language Weaving - Compilers

Once the source code has been woven the resulting ACUs must be compiled to create assemblies. These ACUs now would have a series of references between each other. A reference in terms of compilation can be seen as dependency between ACUs. For example, if ACU A references ACU B, then B must be compiled before A, as the assembly which results from the compilation of B is used in the compilation of A. Thus in an environment where there may be many compilation units, there must be some ordering of compilation.

In addition, the developer may wish to see the source code in its woven state. This would mean that we would have to generate the source code back to the language it was originally expressed in. This

conversion of AST back to its original language is also important for debugging. This is because the code generated from the AST should be added to the assembly at compilation to allow for the source code debugging. To provide the developer with a consistent view of their woven source code at debug-time the source code generated from the AST should be added in the language in which it was originally expressed.

To ensure both correct ordering of compilation and correct conversion of the AST back to the language in which it was originally expressed a Compiler Manager has been integrated into the design to deal with these concerns. The Compiler Manager utilises compilers for different languages to return the ACUs back to their original language and also compile the ACU with the correct debug source code. The Compiler Manager also orders the compilation such that a correct sequence of compilation is followed.

The standardisation that VS offers is also useful when compiling ACU's to assemblies. Each project has as an associated output path which can be extended to store woven assemblies. Thus when compiling source code to an assembly, the references to other assemblies is easily done as woven assemblies will always be stored in a particular area.

The output of a weaving source code then should be a series of assemblies created through the compilation of the compile units and the source files containing the woven source which are of file type for the language that the original source files. The assemblies should also be package with the woven source code for debugging.

3.4.1.4 Application Executing

Once the woven assemblies have been created then the application should be executed automatically. A Runner component has been added to the design to achieve this. This is again is aimed at RAD in the automated execution of an application such that the developer does not have to find the executable file and manually run the application.

3.5 Summary

In this chapter we have looked at the requirements for this project. We present the AspectC# design as a template for the design of a source code weaver. Finally we present the design for SourceWeave.Net through presentation of the issues the design would have to overcome in the to achieve the aims of this work.

Chapter 4

Implementation

4.1 Introduction

In this chapter we explain how the components and models of SourceWeave.Net described in Chapter 3 are physically designed and implemented.

Figure 4.3 illustrates an the physical design of SourceWeave.Net. This diagram has been separated into different sections to represent the system components. In the following sections we take each component of the system and describe how it is designed, implemented and each is utilised to achieve the aims of this work.

4.2 Weaver plugin

SourceWeave.Net has been designed as a plugin to Visual Studio.Net (VS). This was done for many reasons some of which have been briefly discussed in Chapter 3. The many benefits derived from IDE integration are however mainly implementation centric and as such will be explored here.

VS is described, by Microsoft, as a next generation development environment. This environment has been designed to be extensible for the addition of developer tools. The creation of a source code weaver as a plugin is possibly a stretch of the extensibility goal; however, this extensibility facility has been the basis for the implementation of SourceWeave.net. VS integration provides a number of interfaces that give access to the developer environment. SourceWeave.Net takes advantage of the a subset of these features within the environment. Additionally the environment also acts as a managed container, where the addin life-cycle is managed.

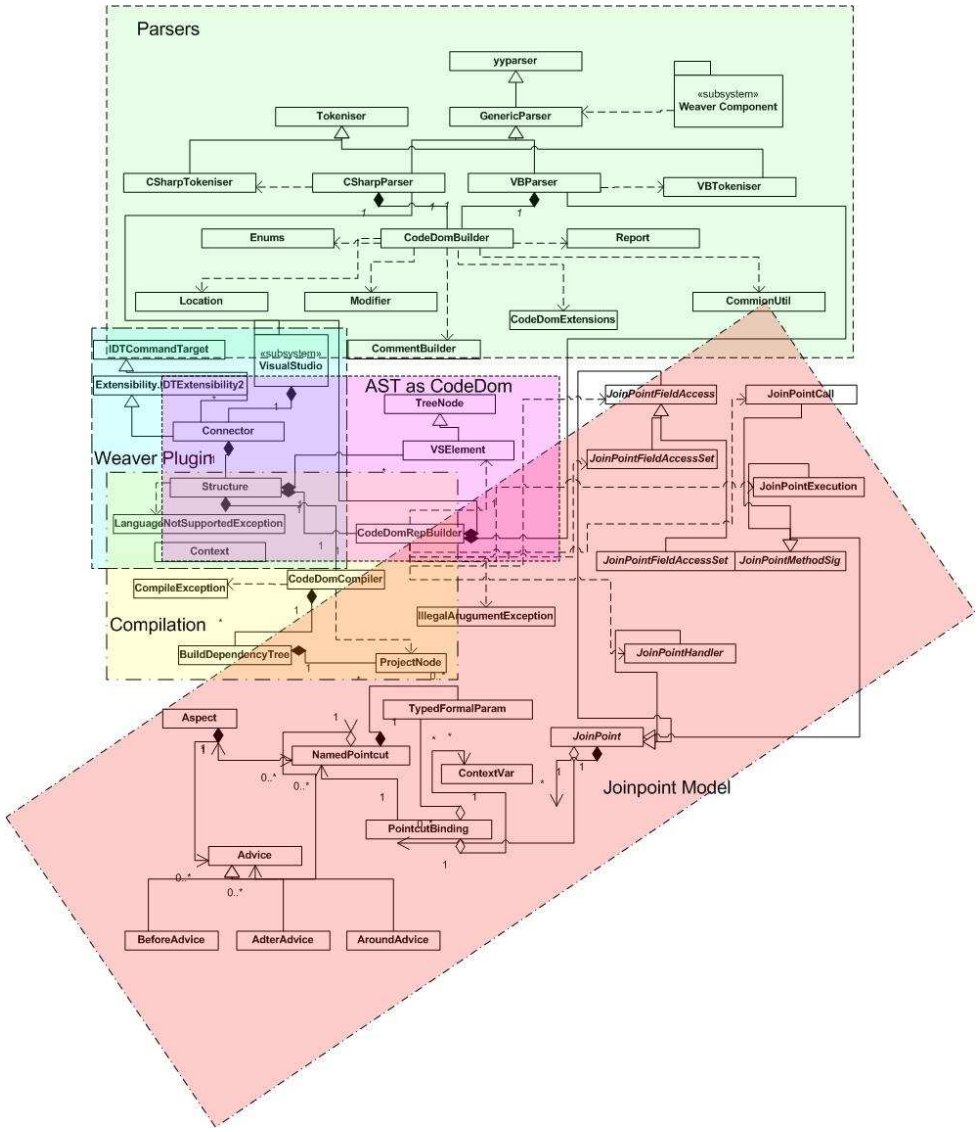

Figure 4.1: SourceWeave.Net Physical Design

4.2.1 Input detection

4.2.1.1 Source Code Organisation and Navigation.

The developer environment enforces a structure to organise source code. A developer who wishes to develop an application will firstly have to begin by creating a solution. Within a solution a developer can then create projects, which can be in different languages, and can have certain different types, for example, a console application or a class library. Within projects there can exist files, wherein source code is stored as part of a project. This standardisation of source organisation is augmented, in that the VS development environment exposes a set of interfaces, for both navigating and retrieving information about the actual source item. An example of this standardisation is presented in Figure

4.2. Through this picture of source code organisation in VS we can begin to understand that to exist in this development environment these items must be represented by some form of structure within the development environment.

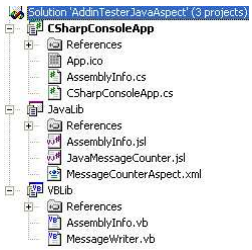

Figure 4.2: Source Code Organisation

4.2.1.2 Visual Studio Automation Object Model

In Figure 4.3 we can see the full range of interfaces that are supported in what is called the Visual Studio Automation Object Model (VSA). The interfaces allow the interaction with the application being developed within the environment. These also provide a view of the base tools that are integrated into the environment, and also the artifacts that can be manipulated within the environment.

4.2.1.3 Utilisation of VSA and Implicit Standardisation in Detection

Through this standardisation and object representation of source code artifacts, the plugin tool can utilise this configuration information to detect inputs by beginning at the solution and moving down through the source organisational tree, discovering both source code files and also the XML aspect descriptor files. In the implementation there certain assumptions made, for example we assume that all source files exist at the level below a project in the organisational tree.

In the implementation, there are two passes through the source tree to discover inputs. The first moves from solution, to project, to project item in search of files named <anyName>Aspect.XML files. All files are detected and references to the VSA objects that represent them are stored. Once all <anyName>Aspect.XML files have been discovered, another descent down the organisational tree from the solution VSA object begins, in the detection of source code. All searches are done in a depth first manner. We followed the search progress to a project item level where source code files are found. The VSA model does continue past this point, but this is an area that will be explored later, in detail, in Section 4.4. However, as a brief introduction to this deeper search, from a project Item level the search continues, to discover the signatures of the namespaces/classes contained within those source files. This signature information is also represented by VSA objects.

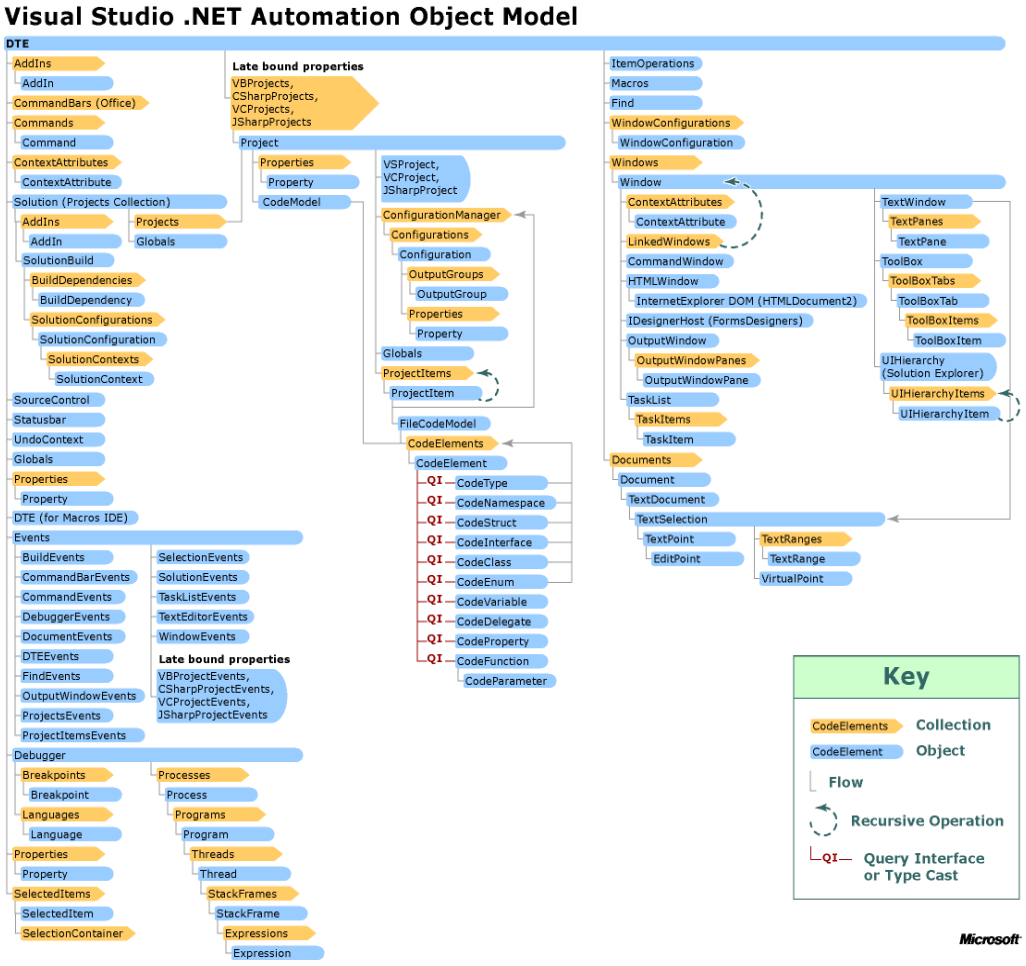

Figure 4.3: VSA Model [Source: Microsoft]

4.2.2 Output Windows

In this implementation there was a need to provide the developer with information about what the tool was doing. We wanted this information to be interactive in that clicking on information would bring the developer to the application fragment from which the information was generated. The ability, for example, to point and click on errors or warnings that arise during weaving,where the error message would bring the user directly to the woven code, would greatly increase the value of the tool to developers. This automation we believe is in keeping with the RAD that VS and .Net have been designed to achieve. In Figure 4.4, there are a number of output windows, where information pertaining to compilation, weaving, joinpoint detection etc. is presented to a developer. There are two main classes of output windows; generic output windows, where information is reduced to a base level and verbose debug windows, where a dump of all information generated is presented. This again is implemented through use of the VSA model which provides interfaces to create output windows.

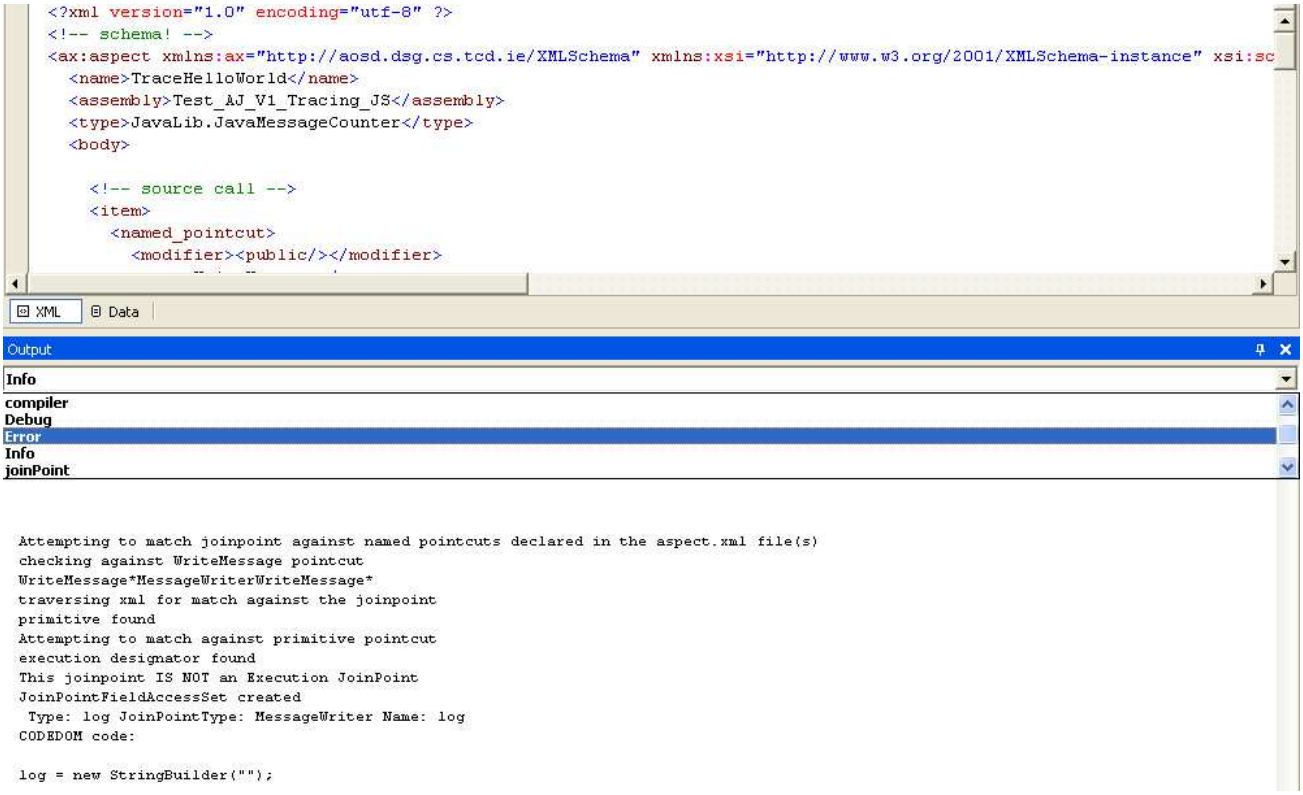

Figure 4.4: Output Windows

4.2.3 Output Creation

The standardisation of resources used within a solution is important for input detection but is equally important for output creation. Having a standard directory structure for output, and programmatic access to this, through the VSA, to access this directory structure is useful, as we can create directories for output of both generated source code files and also woven assemblies. This standardisation is also useful when explicit references between woven assemblies need to be created during compilation, as we need to know the location of assemblies to provide a reference to them. Through extending the formal structure specified by Visual Studio.Net we are maintaining a standard; this removes the need to look for the assemblies at assembly-link or reference-time.

In Figure 4.5 the directory structure created by VS and extended by the SourceWeave.Net plugin is presented. Here we can see that compiled output is placed in a debug folder. Woven assemblies are put in a "weave" folder created by SourceWeave.Net. If a developer however changes the output path, this is not a problem as the tool, through VSA interfaces representing the project, can query the project object representation for the output path of the project.

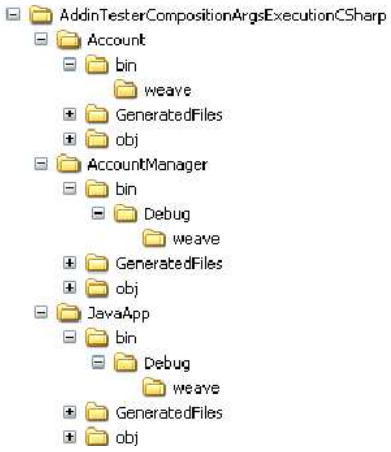

Figure 4.5: Output Standardisation

4.2.4 Design

Now that we have seen how the weaver utilises the Visual Studio environment as a container, in which it benefits from service, access, and standardisation. We turn attention toward the physical design and see how this logical integration is physically achieved.

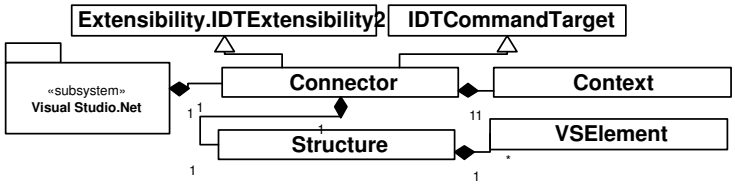

Figure 4.6: Vistual Studio Integration Design

In Figure 4.6, we see that to integrate with VS two interfaces must be implemented. Within these interfaces, methods that allow the integration of the plugin tool into the development environment exist. The Connector class implements these methods, and acts as a management facade between the weaver plugin and the environment. The Connector delegates to the Structure class. Structure is the main class in the system and the entire weaving process is controlled by an instantiation of this class. The Context class is instantiated upon creation of Structure and is a globally shared (through instance propagation) object in which globally required objects and functions are encapsulated.

As we can see in Figure 4.7 the context is composed of OutWindows: these classes represent the windows output windows described in Section 4.2.2. The DTE is a hook into the services provided by the VS environment. The Aspect class represents the holder of information read in from the Aspect XML Descriptor.

We can see in the usage of the VSA model in input detection in Figure 4.8. In this diagram we can see

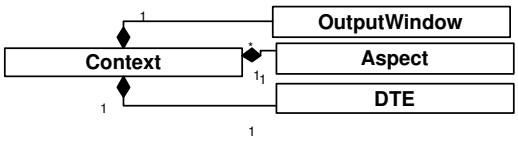

Figure 4.7: ApplicationContext

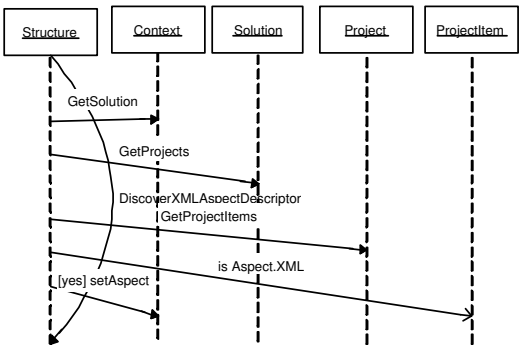

Figure 4.8: Aspect XML Discovery Using VSA Objects

how the VS source code organisation structures solution, project and project item are used to locate the Aspect XML Descriptor. This sequence is also utilised in the source code detection. But in source code detection, wrapper objects of type VSElement are created around the elements detected, these wrappers provide structure as well as encapsulate the VSA source organisational representations of, and provide references to, parallel AST representations of the elements. This also shows that this is a SAX type parsing effect as part of this discovery continues down into the code signatures as is explained in Section 4.4.

4.3 AST as CodeDOM

From the logical design, as presented in Figure 3.4, it is obvious that the need to represent source code as an AST is a critical concern of the implementation. CodeDOM is a namespace in the .Net Class Library, first utilised in AspectC# to represent source code in an abstracted way, and is also exploited by this implementation.

4.3.1 CodeDOM

4.3.1.1 System.CodeDom

The System.CodeDOM [24] namespace contains classes that can be used to represent the elements and structure of a source code document. These classes can be used to model the structure of a source code

document which is malleable, in that at run-time an object representation is created which can be easily altered and amended.

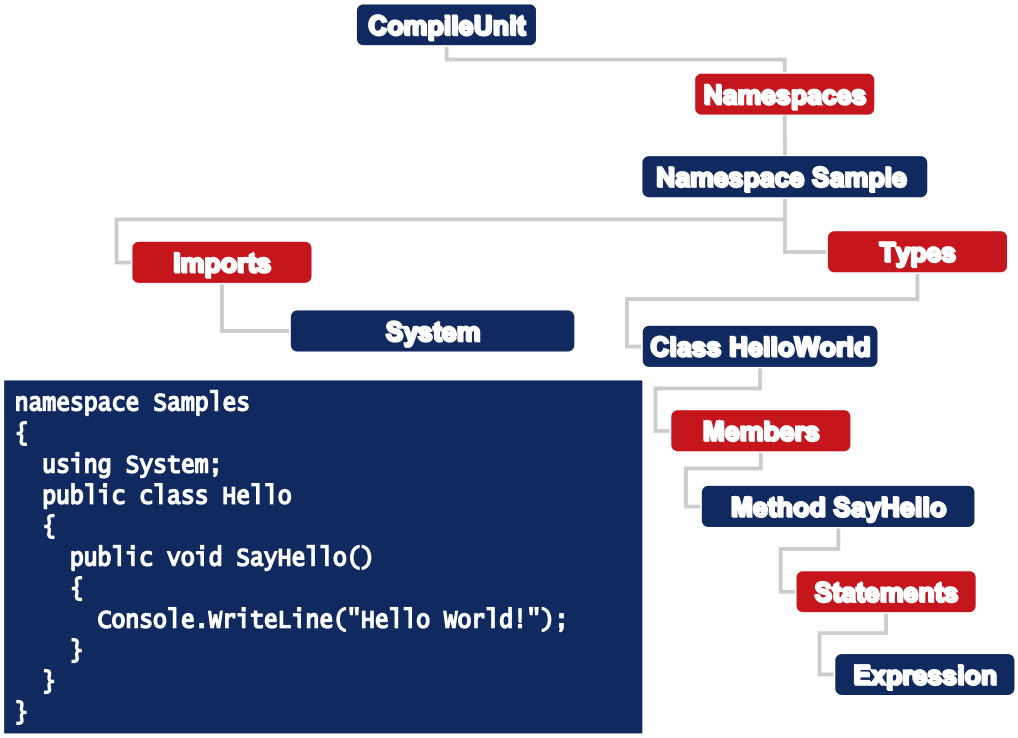

Figure 4.9: CodeDom AST [Source: Microsoft]

In Figure 4.9, we see an example of how the classes within the CodeDOM namespace are used to model source code. We can see the containing type is a *CompileUnit*, this is a unit of abstracted source code that may be compiled. Within this compile unit there is a *Namespace* type to represent the **Sample** namespace this is held within a namespaces type. This pattern is continued down to the the representation of the **Console.WriteLine("Hello Wold")** line of source code as a *Statement* type. This language-independent representation of source code as a object graph model in-memory is central to this implementation as it provides the flexibility needed to weave language differentiated source code.

There are many limitations with CodeDOM model, as it does not provide the classes to abstract all code constructs that are particular to certain languages. For example invariant, precondition and postcondition constructs of the Eiffel language and the module construct of the VBlanguage cannot be abstractly represented in CodeDOM. The exact limitations of CodeDOM to represent source code in all languages is discussed in Chapter 5.

To understand why these limitations exist, one look at how the creators of CodeDOM foresaw its usage. ASP.Net Assembly generation, WSDL proxy generation, Managed C++ WinForms designer base code generation and Visual Studio.Net code wizards are, for example, considered the typical applications areas for CodeDOM. This development focus limits the CodeDOM usage scope to a certain degree.

This idea of an abstract representation of source code as CodeDOM is axial to the implementation of a source code weaver. However it is a poor construct without tools to convert it back into some form of source code, or without some mechanism to compile this language-independent source code representation. The System.CodeDomCompiler namespace provides this tool set.

4.3.1.2 System.CodeDom.Compiler

The System.CodeDom.Compiler [17] namespace contains types for managing the generation and compilation of source code in supported programming languages.

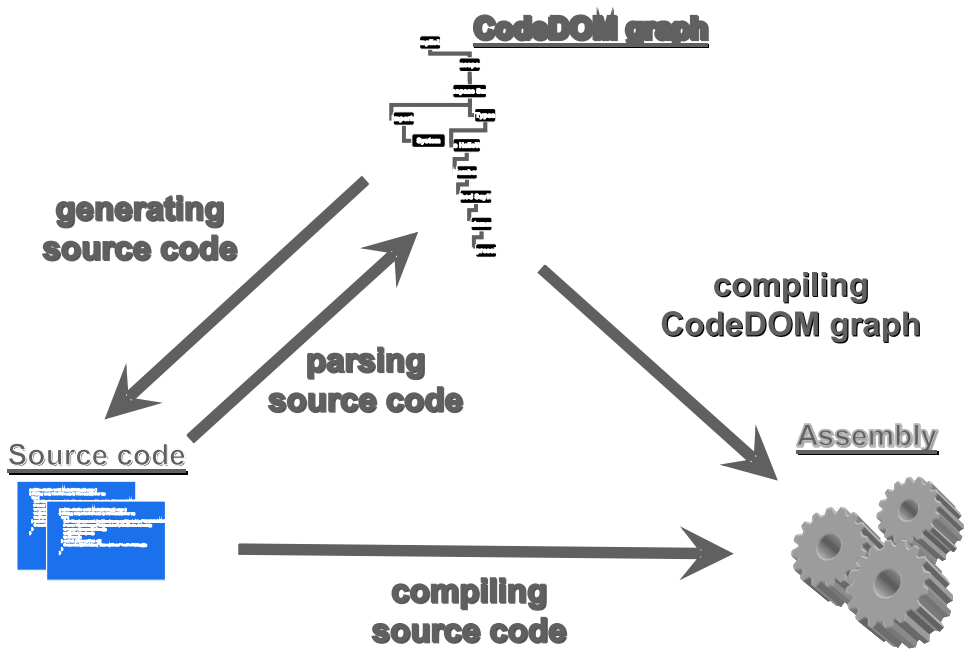

Figure 4.10: CodeDom Tools [Source: Microsoft]

From the diagram we can see that this namespace has been engineered to allow the transition from source code to CodeDOM through source provision of a parsing API. The namespace supports generation of source code from a CodeDOM object graph. The namespace also supports the compilation, of source code, and of a CodeDOM object graph to an assembly.

The .Net framework has been designed to allow multiple languages exist within the framework.

- The questions then are how these languages are all converted to CodeDOM?
- how CodeDOM is converted into a particular language?

- and how is a CodeDOM graph or source code compiled given compilers are designed for a given language?

The answers lie in an abstraction provided by through an abstract factory pattern.

Each language implements the given interfaces to perform the tasks mentioned above. For example, ICodeCompiler and ICodeGenerator are interfaces to compile and generate source code. The abstract factory called an CodeDomProvider is responsible for creating instances of these interfaces. There are providers written for the languages that support CodeDOM. Microsoft.CSharp.CSharpCodeProvider is a type that acts as a concrete factory for the C# language. Here we provide and example of how one would begin to generate C# source code from a CodeDOM graph. Other operations such as code parsing or compilation follows the same pattern.

```
CodeDomProvider csCodeDomProvider = new Microsoft.CSharp.CSharpCodeProvider();
ICodeGenerator csCodeGenerator = csCodeDomProvider.CreateGenerator();
csCodeGenerator.GenerateCodeFromCompileUnit(compileUnit,.....);
```

4.3.2 Partial Implementation for Parsing

Currently, most of the functionality the namespace has contracted to offer, through the defined interfaces, has been implemented. The implementation remains incomplete, in that there is as of yet no parser support. Again we will use another example to illustrate this.

```
CodeDomProvider csCodeDomProvider = new Microsoft.CSharp.CSharpCodeProvider();
ICodeParser csCodeParser = csCodeDomProvider.CreateParser();
csCodeParser.Parse(csSourceCode); <= Exception thrown as csCodeParser is null
```

As we see here there, null, instead of a C# code parser, is returned when we attempt to generate a parser, whereby we can covert source documents into CodeDOM object graphs. There is only one language that has implemented the ICodeParser interface. This language is Managed C++ however there are problems here in that this parser is implemented for a single context, i.e. within the realm of source code generation for the C++ GUI designer. This context coupling means that this parser cannot be utilised in this work.

4.3.3 CodeDom Namespaces usage in Design

As we can see from the logical design of the SourceWeave.Net tool in a physical design this namespace is a core namespace, through which the weaver can take input, create an abstract representation and then use the code generation and compilers as key parts of the weaver.

4.4 Parsers

4.4.1 Source to CodeDOM

The goal of parsing in this context is to create a CodeDOM AST from source code input. As we have noted in Section 4.3.2 the System.CodeDom.Compiler namespace lacks the parsing implementation it has contracted to offer in its interface design. This means for this project there would have to be an implementation of parsers for the different languages that were going to be used in the tool.

4.4.2 AspectC# parsing

AspectC# was as dependent on parsing to CodeDOM. The goal of the AspectC# project was not as dependent on parsing as SourceWeave.Net. One aim of this work is to support an extended joinpoint model, this meant that parsing to an expression level was important. AspectC# represented source code to signature level as it was only concerned with an execution joinpoint and thus only required parsing to this level to discover joinpoints. SourceWeave.Net needed support the identification of an extensive set of joinpoints. Thus parsing to a a level where these can be discovered is crucial.

AspectC# was also focused on one language, C#, and as such did not need to deal with deep abstraction. Any code beyond signature constructs, were put into literal constructs. In the AspectC# implementation any statement level source code is represented literally in a CodeSnippetStatement CodeDOM representation which is basically an extension of the String type. This was acceptable as the language at that joinpoint didn't matter, as it was all C#. In this implementation, the goal of cross-language weaving would not allow such literal code expression. In this work an absolute code abstraction was implemented, to allow code be woven regardless of language.

4.4.3 Language Selection

As this was a proof of concept implementation and was heavily time constrained, three languages were chosen for parsing: C#,J# and VB.Net. The introduction of Eiffel and C++ was attempted, but because of time constraints parser support for these languages was not completed before the writing of this thesis.

4.4.4 Language Support and Extensibility

This tool can support any language that conforms to the .Net language specifications. The only constraint is that the language must support CodeDOM, through creation of a CodeDomProvider and implementing the required interfaces. There is no work to be done outside of creation of a parser from the language source code to CodeDOM beyond this. To reiterate this point, no third party work needs to be done to add a new language to this tool beyond the work to implement the language on the .Net framework.

4.4.5 VSA as a Basis for Parsing

In section 4.2.1.3 we looked how the VSA model was being utilised in the physical design in terms of input detection. The VSA model is further utilised in terms of parsing the source code input. The VSA model allows the extraction of source code information to a signature level. VSA provides us with a set of interfaces to abstractly represent source code. In this way we can use the Common Environment Object Model (CEOM), a subset of VSA, to represent source level signatures as objects. For example, we can see in Figure 4.3 there are a number of interfaces such as CodeClass, CodeFunction and CodeParameter which can be used to extract information about source code constructs in a language-independent way. Thus, the VSA provides a language-independent way to parse source code written in different languages through interrogation objects that represent source level language-independent signature based information. From the exposed information we can then create a CodeDOM model to a signature level. What remains, however, is the internal behaviour of the methods or the initialisation code that is attached through assignment to fields. To parse this degree we are constrained by the VSA model as there is no support for the retrieval of objects that represent information on a statement or expression level. Thus we are forced to introduce a new model beyond VSA to achieve parsing to the base grammar level.

4.4.6 Statement level Parsers

Because the VSA model could not provide the language-independent parsing of source to a completely abstract level, this meant that we had to provide parsers that would create a complete abstraction of source code in order to provide the joinpoint discovery and weaving facility, which is described in section 4.4.2. For parsing it was decided that we would not attempt to create a parser from scratch, we would attempt to use existing parsers. There are no full implementations of parsers that fully convert source code to CodeDOM.

Ivan Zderadicka had partially implemented a C# to CodeDOM [10] parser. This parser is a port of a C# parser which was written as part of the Mono project's [16] C# compiler. To implement a C# to CodeDOM parser, we utilised the work begun on the C# to CodeDOM parser to fulfil to the level of parsing that SourceWeave.Net required. The Mono project also had a partially implemented VB compiler, a component of which is a partially implemented VB parser. These parsers would be the basis for the creation of full language (with respect to this project) parsers for C# and VB.Net to CodeDOM. These components were developed as separate components and integrated into the SourceWeave.Net tool for reuse purposes.

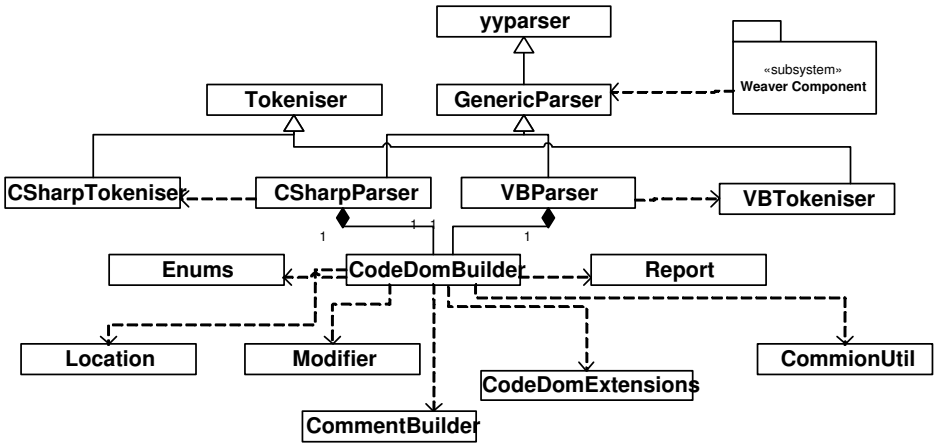

Figure 4.11: Parser Components Archietcture

4.4.6.1 Parser Design

In Figure 4.11 we explore the parser design. Here we see that the parsers themselves are different and use different tokenisers but both the C# and VB parsers utilise the same infrastructure in the creation of CodeDOM representation of source code. Both extend the GenericParser, thus through the GenericParser interface the different parsers can be accessed. This shows that by utilising the same infrastructure and extending the GenericParser more language parsers can be added to this parsing component in future extensions of the language base supported by SourceWeave.Net.

4.4.6.2 Extending the CodeDom Compiler Namespace.

The creation of these parsers now means that there is a possibility of implementing the ICodeParser. This would be useful in a both a redesign of this work for further research , but also to go beyond this work in other areas.

4.4.7 VSA and Language Parser Integration

For the VSA based parsing mechanism and parsers created, a degree of integration had to be achieved to link the CodeDOM structures created by both so that they could be joined at the appropriate level. To maintain the parsers as separate components we chose to implement finder methods to find the parsed CodeDOM in the tree and then point a reference toward this part of the parser created CodeDOM tree.

Figure 4.12 illustrates the end result of parsing C# source code with a VSA and C# parser. The source code is written in C# and thus the VSA parser which is language-independent parses the signature constructs to CodeDOM, but the C# parser then parses the statement level source code to CodeDOM. However we can see in figure 4.13, that the objects of type VSElement provide a structure which mimics the VSA element's structure such that CodeDOM can be organised within and extend this structure.

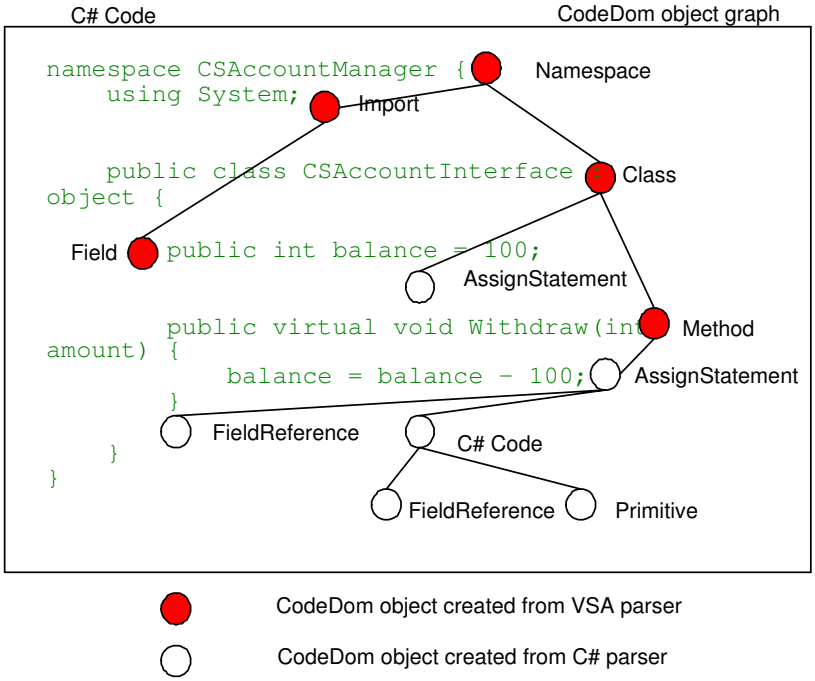

Figure 4.12: VSA and C Sharp Parser integration to create CodeDom Object Graph

4.4.8 Source Organisation to CodeDOM Organisation

The VSA model as described in section 4.2.1.3, provides a level of hierarchical structure to organise source code. After parsing there are a number of CodeDOM source code representations, these must be organised into a well-defined structure, for both weaving and compilation. As the VSA model provides a well-defined structure the CodeDOM representations; have been mapped to this structure. Figure 4.13 shows this structuring. The objects of type VSElement mimic the VSA structure and provide links to the CodeDOM object graph created. Thus during weaving or compilation CodeDOM objects can be found through navigation of the VSElement objects.

4.5 Joinpoint Model

4.5.1 Overview

The joinpoint model is at the core of SourceWeave.Net. The joinpoint model's function is to identify joinpoints in the source code, where additional crosscutting behaviour should be introduced at those joinpoints, then weave that crosscutting behaviour into the source code which is represented by the joinpoint. The logical design of the joinpoint model presented in section 3.4.1.2 is separated into three models, a mapping, a weaving and an instantiation model. These logically distinct models however have been implemented within a single physical design illustrated in Figure 4.14. This class structure

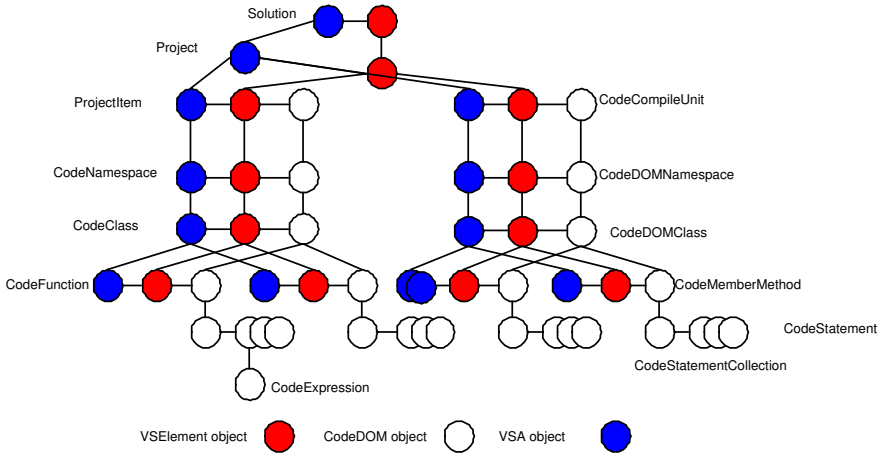

Figure 4.13: Source and CodeDOM representation object structure

represents the XML aspect specification in Section 1 of Figure 4.14. Section 2 of the diagram represents the source perspective where joinpoints are discovered. Section 3 of the diagram represents the link between the source and XML aspect specification.

We will describe the how the models are implemented within this structure in the following sections.

4.5.2 Mapping Model

This weaver can identify joinpoints in source code, and distinguish which joinpoints are points that a developer wishes to crosscut with additional behaviour. The mapping model has been implemented to support this identification process.

There are two different parsing structures incorporated in this tool. The joinpoint discovery has been implemented in differing ways because of this. The VSA parser implementation allows detection of joinpoints as source parsing to CodeDOM is done. The implemented C# and VB parsers do not discover joinpoints as they are parsed. A policy of post parse CodeDOM analysis is employed to detect joinpoints when utilising these parsers to statement level.

In either joinpoint detection situation the mapping model is used to match the source code constructs against the pointcuts declared in XML aspect characterisation.

4.5.2.1 Matching Sequence

In this section we describe the sequence of steps implemented in SourceWeave.Net to identify joinpoints. These joinpoints have been declared as points within source code at which crosscutting behaviour should be woven into the source code represented by the joinpoint. This is a high level description, the details of which will be explained in the following sections.

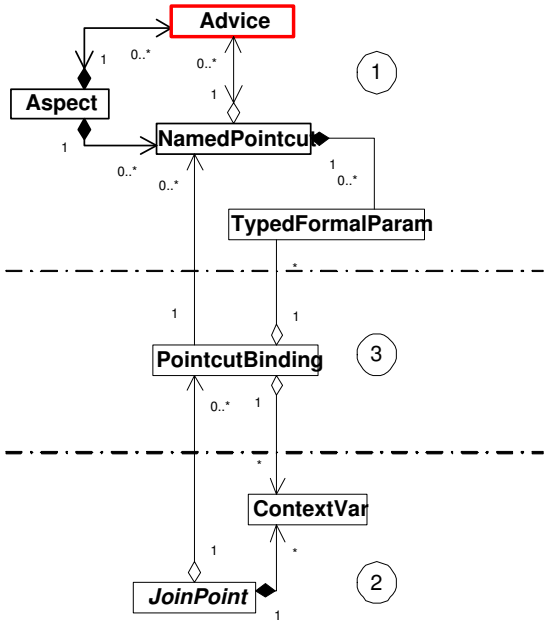

Figure 4.14: Joinpoint model[Source: Weave.Net]

We begin at the input detection phase. At this point the XML files that contain the declarative information about an aspect are read and a set of objects are created to represent this information. These include an Aspect object which represents information declared to describe the aspect, NamedPointcut objects which represent information declared to describe the pointcuts and Advice objects which represent information declared to describe Advice. Once these structures are in place, then the joinpoints can be matched against these patterns modelled as NamedPointcut objects.

After this has been done the source files are then parsed. During parsing, a source code pattern is identified as being a possible joinpoint. The properties of the source code are then used to create a Joinpoint object to represent that source code pattern. Then the parser begins to attempt to match against the pointcuts held within the Aspect objects. The Joinpoint object is sent to the Aspect object which in turn attempts to match the joinpoint against the NamedPointcuts objects that it contains.

A NamedPointcut object cut may represent a primitive pointcut or a composition of primitive pointcuts. Thus there are two scenarios that must be dealt with, matching against a primitive pointcut and matching against a composition of primitive pointcuts. In the first case where the pointcut is a primitive pointcut, the properties of the source code are matched against the patterns declared in the primitive pointcut. If the source code properties match, then a reference to the joinpoint object is stored in a list of joinpoints that are to be crosscut with additional behaviour. The composite case is more complex, here the

composite is recursively broken down to a sequence of primitive pointcuts to be matched against. If the Joinpoint object matches all the primitive pointcuts in a way that the logical connectors of that determine the relationship between the primitive pointcuts within the composition, then a reference to the joinpoint object will be added to the list of joinpoints to be woven. Also when the Joinpoint object matches a NamedPointcut object a PointcutBinding object is created. This is an object that holds a reference to the NamedPointcut. A reference to this object is added to a collection within the joinpoint object. Thus at this point there is a reference in the joinpoint to a collection of PointcutBinding objects which in turn hold references to NamedPointcut objects which hold references to the advice objects. The advice objects encapsulate information about the crosscutting behaviour that is to crosscut the source code at the joinpoint as well as information about how the crosscutting behaviour is to crosscut the source code at the joinpoint.

This sequence is followed for every possible joinpoint discovered when parsing the source code. When all source code has been parsed then weaving of crosscutting behaviour at those joinpoints may begin.

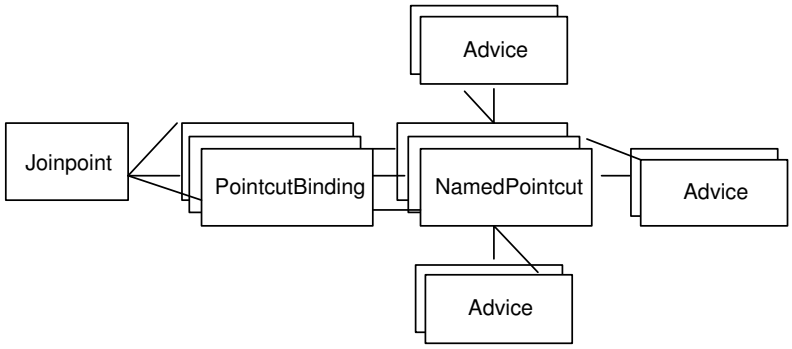

Figure 4.15: Object Struture After Mapping

Figure 4.15 illustrates the object relationship after a joinpoint has matched three pointcuts each of which have two advice declarations associated with the pointcuts. This model shows that a joinpoint, can after mapping occurs through the references formed during mapping, access the advice information needed during weaving.

4.5.2.2 Joinpoint Design

As described in section 2.3.1 a joinpoint is a well-defined point of execution in the program. In section 2 of figure 4.14 we see the joinpoints in the context of the overall joinpoint model. We see here that joinpoints exist in the context of source code traversal and examination.

Because SourceWeave.Net is source code based, this execution point is represented by source code. Source code can be thought of as an instruction set for execution. Thus we can identify execution of behaviour through analysing source code. A joinpoint is representation of source code; source code in turn is a

representation of a point of execution. In this way we can identify points of execution within source code and represent them as joinpoints.

For this reason a joinpoint is designed to encapsulate the properties and characteristics of the source code that the joinpoint represents. Additionally access to this information is part of the joinpoint design. This design has been created to facilitate matching of joinpoints against pointcuts.

There are many types of joinpoint supported by SourceWeave.Net. These joinpoint types include exception, call, set, get and handler joinpoints. All of these joinpoints represent differing types of source code patterns. An execution joinpoint, for example, encapsulates the method signature information. This is because an execution joinpoint is meant to represent the execution of the behaviour of a method body within the execution of a system. Thus, when matching an execution joinpoint against an execution pointcut, the joinpoint object can be queried about the method information to see if this method information matches what is declared within the pointcut.

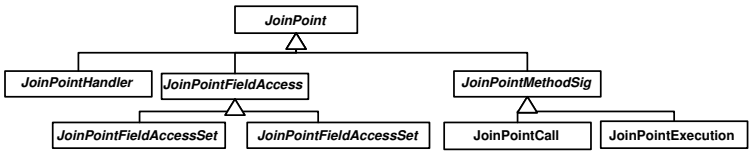

Figure 4.16: Joinpoints

Figure 4.16 presents the structure of the implemented joinpoints. The actual joinpoint types are represented through creation of classes which represent these joinpoint types. Because certain joinpoints encapsulate similar information, for example the set and get joinpoints encapsulate information about a field, inheritance is leveraged in these instances to capture these similarities. As such, we can see from the diagram that JoinPointFieldAccessSet represents a set joinpoint and JoinPointFieldAccessGet represents a get joinpoint. However, much of the implemented behaviour to do with mapping is field based, and as such is captured in the JoinPointFieldAccess class. We can see this pattern repeated in the JoinPointExecution and JoinPointCall classes which represent the execution and call joinpoints which inherit from JoinPointMethodSig. JoinPointMethodSig here captures the behaviour needed for matching method based joinpoints. The JoinPointHandler class represents a handler joinpoint, there are no joinpoints implemented in this joinpoint model that need to encapsulate exception information and as such JoinPointHandler inherits directly from the JoinPoint class which encapsulates all the similarities that exist between joinpoints.

Inheritance here allows not only reuse and standardisation for extension of the supported joinpoints but the advantages of polymorphism can also be leveraged. Thus all joinpoints can be treated uniformally where necessary and then be specialised based on the matching context.

4.5.2.3 Joinpoint Creation

Joinpoints are created at points where well-defined source code patterns exist. They are populated with information about the pattern which triggered initialisation of the joinpoint and the location of the pattern in the source code tree. The pattern information is accessible through accessors on the joinpoint's interface for matching against pointcut declarations. A joinpoint also acts as a marker, for weaving in that it records the location where the joinpoint was discovered. Beyond being a marker a joinpoint also holds weaving functionality. This will also be later explained in section 4.5.4.

4.5.3 XML specification Modelled

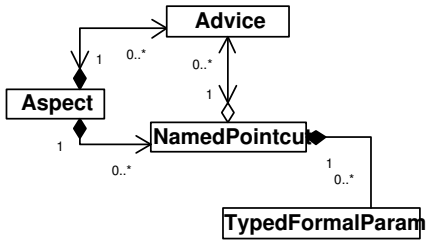

Figure 4.17: XML Specification Modeled

The XML aspect specification[12] enables declaration of the crosscutting characteristics of an aspect in a language-independent way . A developer can thus declare both pointcuts and advice within an XML document. To enable matching of joinpoints against declared pointcuts we implemented a structure which supports the representation XML document as a set of objects. The XML document not only holds information but also expresses a set of complex set of relationships between the declared information which is defined within the XML aspect specification. The structure implemented is presented in Figure 4.17. This class structure also follows the XML structures and relationships defined within XML aspect specification.

Each of these structures uses the System.XML namespace in the .Net class library to read this information. Each structure represents a particular subset of information within the XML document, and as such, each structure is responsible for reading the information which the structure is to encapsulate. The document is detected and a DOM parser is then used to create an object graph to represent the XML. For each XML aspect descriptor detected an Aspect object is created. The information the document holds about the aspect, such as the name of the aspect and the class in which the crosscutting behaviour which the document characterises, is stored in the aspect. The pointcut and advice XML elements are the passed to used to create instances of the NamedPointcut and Advice classes.

Figure 4.18 illustrates this conversion of the XML information to an object structure. Each object in the diagram is a representation of the information held within the XML document and the relationships

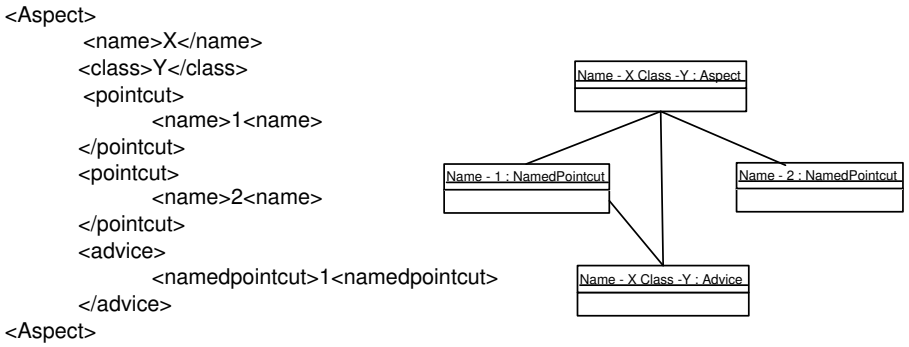

Figure 4.18: XML declaration to Object Representation

between the XML elements is expressed in the object references. The XML in the diagram represents the structures within the XML aspect descriptor file. As we can see from the diagram this structure is recreated in an object model.

4.5.4 Weaving Model

4.5.4.1 Pre Weave

Once the entire source base has been parsed and all joinpoints discovered in the source have been matched against named pointcuts, we are left with a collection of matched joinpoints which represent source code fragments at which crosscutting advice behaviour should be woven.

4.5.4.2 Advice

Figure 4.19: Advice model

As described in section 4.5.3 advice is declared in the XML aspect descriptor by the developer. The developer declares methods as being crosscutting behaviour as well as associating the advice with a pointcut. Thus when a pointcut is matched with a joinpoint, the advice which is associated with the pointcut is then the behaviour that is to crosscut source code represented by the joinpoint.

Three types of advice have been implemented corresponding to three types of advice defined in the XML aspect specification. Figure 4.19 shows the advice structures that have been implemented to represent these advice types. AdviceBefore, AdviceAround and AdviceAfter represent the three advice types. All advice encapsulates the similar information declared in the advice XML elements. Thus, these similarities

are then implemented in an Advice class which is extended by the AdviceBefore ,AdviceAround and AdviceAfter classes. Thus when advice is being woven at joinpoints, the type of the advice is denoted by the type of the advice object.

4.5.4.3 Joinpoints in weaving

As discussed in section 4.5.1, there is a major overlap in the implementation of the mapping,weaving and instantiation models. Joinpoints are a good example of this: the JoinPoint class defines in its interface three methods AddBeforeAdvice, AddAroundAdvice and AddAfterAdvice. Since the location of the source code which is to be crosscut is easily encapsulated in this type, it makes sense that weaving functionality should also be included here to take advantage of this information for weaving purposes. The inheritance structure is also suited to weaving. Each joinpoint type requires a different form of weaving based in the type of the joinpoint. For example an execution joinpoint will introduce crosscutting behaviour before, after or around the source code within the method body whereas a handler joinpoint will only implement before advice introduction as it would not make sense to allow after advice as this would be unreachable code. In this way each joinpoint has the chance to specialise the way that it weaves advice into the base location it targets.

4.5.4.4 Discovering Advice

From Figures 4.14 and 4.15 we see there is a link established during the matching process that allows the joinpoint access the advice elements that are to be woven into the code at that joinpoint. When for example the AddBeforeAdvice method is invoked in a JoinPoint instance, the joinpoint inspects its pointcut bindings to assess which named pointcuts it matched and then looks for the before advice, if any, that is associated with that joinpoint. Once an instance of the Advice class is found the instance is interrogated to see what method should be called on the aspect. Once this is done a search for the actual advice method signature is carried out and the CodeDOM advice representation is retrieved.

4.5.4.5 Instantiation Model - Introducing crosscutting behaviour to Code

As has been pointed out in chapter 3 there has been a need to ensure declarative completeness pre and post weave. In this implementation an instantiation model has been implemented. Under this model an instance of an aspect is created under a certain scope and then the advice method exposed by that aspect is invoked to add that behaviour. Aspect Instantiation is done at a method level in this model i.e. for each method there is only one Aspect instantiated. This is a model that is extensible to having an instantiation per advice invocation and singleton instantiation such that the aspect has application wide context. Class scope aspect instantiation could have been used but as weaving effects a method's behaviour, it was deemed that method scoped instantiation would be more effective as this would demonstrate which

methods were being crosscut by which aspects more clearly. Per advice instantiation would have made woven code very cluttered and singleton could have easily been implemented but for time constraints.

4.5.4.6 Adding source declaration

When we have woven a type of class, the namespace where this class exists must also be woven into the code. The idea of an aspect is the separation of concerns. This means that in code the aspect type depending on its namespace membership may or may not be declared as using this set of types. Thus, when weaving under an instantiation model we must also take into account the declaration of the type's containing namespace being used in the base code where weaving and thus type introduction occurs.

4.5.4.7 Dynamic reflective joinpoint access

This implementation supports reflective access to joinpoint information. A developer can declare a parameter of type ThisJoinPoint in an advice method signature that encapsulates crosscutting behaviour. If the weaver sees this then in the call to the advice method the relevant source code is injected into the base code to allow dynamic reflective access to joinpoint information. Thus, the developer can in the advice behaviour query the ThisJoinPoint parameter for information about the joinpoint that, when executing, is calling the advice behaviour. Thus at run-time a system can perform different information based the joinpoint that is executing.

4.5.4.8 Context

This implementation supports the idea of sharing context bound information between base code and aspect such that structures being used in the execution of base code can be shared with aspect behaviour. This information is also exposed to the advice method through method signature arguments. The woven code advice execution takes the source identifier of some sort that represents the context structure to be passed to the advice method by adding the reference to the parameters of the advice method call. Primitive pointcuts including args, this and target allow the passing of context. These pointcuts are used as a generic match against all source traversal generated joinpoints such as call/execution/set/get joinpoints. Thus there is no implementation of a JoinPointThis, JoinPointTarget or JoinPointArgs. Instead these ideas are encapsulated in the JoinPoint class as fields that are populated upon Joinpoint creation. These fields are instances of the ContextVar type, or in the case of the args, an array of ContextVar instances.

4.5.4.9 Weaving dependencies

When we weave source code in an environment where source code is organised into compilation units that are language centric, we must then not only weave source code but also take care to add references

between code compilation units. So, again taking our example stated earlier in the chapter, if we wish to weave aspect code written in J# into base code written in VB.Net we must first modularise the J# code into one compilation unit and the VB.Net code into another compilation unit. Now if we wish to both create an instance of the J# aspect and invoke an advice method on that instance, we must firstly also create a reference from the VB.Net compilation unit to the J# compilation unit to ensure that VB.Net code can recognise the structures utilised through weaving these structures into the J# base code.

4.6 Compilation

4.6.1 Overview

Once code has been woven, it must then be compiled. Seeing as code is expressed in a neutral way the code could conceptually be converted to the same language and compiled as such. This however does not really add any value to the developer's experience or functionality of the tool. One of the major aims of this project was to focus on ease of use for developers who would use the tool. In the pursuit of this goal the weaver has been implemented to output the woven code in the languages in which they were written, then compile the woven source code in the compiler written for the language of the given source code.

4.6.2 Dependencies

As mentioned in section 4.5.4.9 there are compilation units that are language centric. The reasoning behind this is that different languages utilise different compilers. But however there is a further issue here. These compile units reference each other at a source level to allow the referencer access at a source level the types that a reference holds within the referencee's compile unit. These references at a compilation level are compilation dependencies, that is to say, compilation of one source unit cannot occur before another because it holds types required for the first unit to successfully compile. Thus, before compilation, the dependencies between these units and a compilation plan or schedule must be devised to ensure that compilation of these units occurs correctly.

Here in figure 4.20 we see the structures used to deal with this complexity. The BuildDependencyTree looks at all the references as dependencies and creates a tree of ProjectNodes. These are sorted from the start up project or the project in Visual Studio that will become an exe and moves down the reference tree creating a comparable dependency. Then a topological sort is done to assess which project or compilation unit will be compiled next; once this has been compiled, it is eliminated from the tree and the references to it are deleted and the sort begins again. Thus from this process a clean ordered compilation of all the projects/compile units can take place.

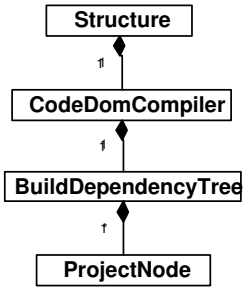

Figure 4.20: Compilation model

4.6.3 Compilers

Each project containing source code in a particular language will utilise a separated compiler. CodeDOM, as discussed in section 4.3.1, provides a structure to allow us to utilise compilers with CodeDOM object graphs. The compilation component has been built around this available tool set (see figure 4.21). Interfaces are used as views over all compilers because then all compilers can be treated similarly and this component can be easily extended to deal with further languages which then only need to add a new CodeDomProvider to be introduced.

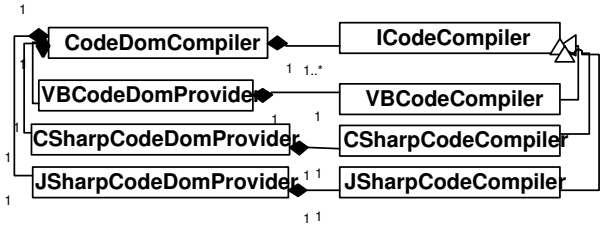

Figure 4.21: Compilers

4.6.4 Output

Compilation outputs assemblies in the form of exe files and dll files. However there is also source code output that is created in the process of compiling but kept. To allow developers track the code generated for each compile a new date stamp named folder is created. In this folder the source files are output. In terms of the exe/dll files there are certain compiler switches turned on at compilation. An important point to note here is the option to add the woven source code to the assembly to later enable debugging of the woven code.

4.6.5 Errors

Errors from the compiler are displayed in the Error output window of the development environment. Here the line, class and compiler error messages are displayed. Upon a click the development environment opens the erroneous code in the editor so that the developer can examine the code.

4.7 Summary

This chapter has looked at the implementation of the weaver. The different components that make up the weaver, how they are physically designed, and what technologies were used to implement them.

Chapter 5

Evaluation

5.1 Introduction

In this chapter we will evaluate SourceWeave.Net's cross-language source code weaving approach as a migration path for the .Net framework to AOSD. We will then evaluate this approach in relation to the other approaches described in Chapter 2. We also examine the limitations of this approach in detail.

5.2 SourceWeave.Net versus the Rest

The .Net framework as discussed in Chapter 2, is a heavily standardised multi-language platform. There are many approaches to introducing AOSD concepts to the .Net framework. These approaches fall into 2 main categories, the first is where a code translation component of the .Net framework is modified and the second is where the .Net framework is extended and extra code translation components are added.

To change the .Net framework two approaches can be taken. AOSD can be introduced either, at run-time through CLR, alteration or at compile-time through language extension and compiler alteration.

In framework extension two approaches can be taken. A code translation point to implement AOSD concepts can be added either pre-compilation-time, which works with source code, or post-compilation-time, which works with IL.

SourceWeave.Net is a pre-compile-time extension of the .Net framework as a cross-language source code weaver. This represents a new approach for the migration to the .Net framework to introduce AOSD concepts.

In the following sections we evaluate SourceWeave.Net against these other approaches under a number of headings that we believe are crucial issues for acceptance as a possible migration path of the .Net framework to AOSD.

5.2.1 Joinpoint Model Support

The joinpoint model determines the level to which concerns can be separated and re-composed into a working system. Thus, any approach that is to be considered as a migration path for the .Net framework to AOSD must support an extensive joinpoint model. In this section we profile the joinpoint models that have been implemented on the .Net framework and position SourceWeave.Net in relation to these implementations.

Many of the IL weaving approaches find supporting a rich joinpoint model difficult as many have no real support for easily accessing IL based information beyond the metadata that is contained within an assembly in which the IL is packaged. Approaches such as AOP# and CLAW (Sections 2.4.5 and 2.4.6) have been abandoned because they could not easily analyse IL code to discover joinpoints. These approaches were implemented as unmanaged COM components and as such did not have access to the .Net class library which implements a rich set of libraries such as the reflection API which allow access to IL packaged in assemblies.On the other hand IL based approaches such as Weave.Net (Section 2.4.4), which is implemented as a managed component, has implemented an extensive joinpoint model. Weave.Net, however, cannot implement the proceed command associated with around advice due to IL limitations. This demonstrates that an IL approach to AOSD.Net can support a rich joinpoint model but cannot support certain constructs.

The run-time approaches such as AspectBuilder (Section 2.4.9) that have been implemented have supported a limited and intrusive joinpoint model. This is because these approaches have followed the composition filters model (Section 2.3.3). Thus, crosscutting behaviour can be only be woven at an interface level. This limits the opportunity to modularise crosscutting behaviour that is encapsulated behind interfaces.

Approaches that extend languages through modify compilers to introduce AOSD constructs into those languages have been shown to support extensive joinpoint models in a single language environment. This is because the joinpoint model can be built around the constructs that a particular language supports, AspectJ (Section 2.3.1) is a case in point on the Java platform. The creators of IlaC# (Section 2.4.2) claim that their extension to the C# language will support an extensive joinpoint model that enables instance level weaving. In the IlaC# approach, however, the ability to weave concerns expressed in different languages is lost in this approach.

SourceWeave.Net has demonstrated that an extensive joinpoint model can be supported through working with source code in a cross-language environment. SourceWeave.Net has implemented a subset of the joinpoint model features of AspectJ which is recognised as having an extensive joinpoint model. The architecture of the joinpoint model that SourceWeave.Net has implemented is however has been designed to be extended to support all of theses features.

SourceWeave.Net as a source code weaver and some IL based weavers can identify joinpoints at an expression level . The run-time AOSD implementations are restricted to supporting only interface based

joinpoints.

Unlike the IL weaving approaches such as Weave.Net, SourceWeave.Net can support constructs such as the proceed command as described above. This is because source code is more expressive than IL and as such is more malleable as a basis for implementing all features of a joinpoint model.

SourceWeave.Net can also be extended in ways that other approaches cannot. For example, SourceWeave.Net could be extended to implement language specific joinpoints. By extending the SourceWeave.Net, constructs particular to certain languages could be crosscut with behaviour. Although this has not been implemented, the architecture of the SourceWeave.Net joinpoint model supports this extension. Only a cross-language source code weaving approach can support this. IL and run-time approaches cannot determine the language specific constructs, because at the level these approaches are working at, artifacts are language-independent. Again language specific approaches can specialise the joinpoint model to the constructs the specific language supports, but these approaches cannot weave cross-language.

5.2.2 Intrusion Level

There are approaches that need to modify base code to make joinpoints explicit. For example, both AspectBuilder (Section 2.4.9) and Loom.Net (Section 2.4.8) need to insert some markers in the base code to explicitly mark joinpoints. This is not scalable, as in a system which is very large, it may not be feasible to alter the base code to add explicit joinpoints.

Most approaches, however, do not follow this intrusive model. Common to many approaches such as CLAW, AOP#, AOP.Net, AspectC# and Weave.Net is the usage of a declarative description of how to identify joinpoints which are to be crosscut. In these implementations the declarative description is read and the artifacts that are inputs to these implementations are analysed to identify joinpoints that match the declarative description.

SourceWeave is a non intrusive approach. Like the approaches mentioned in the previous paragraph, all joinpoint detection is performed dynamically through non-intrusive declaration of the crosscutting characteristics in XML which is read by SourceWeave.Net and used to identify joinpoints in source code written different languages.

As such SourceWeave.Net represents a scalable solution, as the size of the application does not inhibit the ability to identify joinpoints. Scalability has, however, been achieved by many other approaches mentioned above.

5.2.3 Language Support

The .Net framework supports over 20 languages. To support true cross-language weaving all languages on the .Net framework should be included in any approach which is to be considered as a migration path to the .Net framework.

J# is a .Net compliant language that conforms to the CLS, however the output of the J# compiler does not strictly conform to the CLI. Due to this issue, IL based approaches such as Weave.Net cannot weave components written in J#. However, by weaving at the IL level cross-language weaving is implicitly supported as IL is language-independent. Thus, IL weavers can support most of the .Net compliant languages. Only languages such as J# that do not conform to the CLI cannot be supported by current IL weaver implementations such as Weave.Net.

Approaches that modify the run-time, such as AspectBuilder (Section 2.4.9) environment to implement AOSD can support all languages. This is because they work below the IL level. Assemblies generated by the J# compiler, despite not being strictly CLI complaint can execute on the CLR and as such can be crosscut with additional behaviour at run-time.

Approaches such as IlaC# (Section 2.4.2) and Aspect.Net (Section 2.4.3), that extend languages to introduce AOSD constructs through compiler modification. These approaches only allow weaving within the language that is being extended. In both cases the language is C#. Approaches such as AspectC# (Section 2.4.1) which implement single-language source code weaving as a preprocessor to the C# compiler can only weave within the C# language also.

SourceWeave.Net has demonstrated that it can weave crosscutting concerns expressed in VB,C# and J# source code. The use of CodeDOM has shown that the range of languages can be supported by SourceWeave.Net can be extended to encompass the full complement of .Net compliant languages that support CodeDOM.

5.2.4 Framework Alteration

The .Net framework is based on a set of standards and specifications. Approaches that attempt to modify the actual specified framework must change a framework that is very large and complex, has taken much investment to create, and in which multiple contributors are involved.

Working with IL can be seen as working within the framework, as IL is created by the components, within the framework, and as such IL based AOSD implementations are thus more constrained by the framework than source code based AOSD implementations.

By moving outside or above the framework, to work with source code which is the base input to the framework, no change to the framework is needed. SourceWeave.Net demonstrates that AOSD implementation can be achieved through moving outside the framework. We believe that protecting the investment in the .Net framework is a benefit of SourceWeave.Net's approach.

Through non-alteration of the framework to implement AOSD concepts, the facilities the framework provides, that are standards dependent can be retained within the AOSD implementation. However, in changing the framework beyond the standards, the availability of these facilities may be lost within the AOSD implementation. This point is further illustrated in Section 5.2.5.

5.2.5 Debugging

IL based and run-time approaches to implementing AOSD on the .Net framework cannot support the debugging of source code. A connection between source code and IL is created at compile-time to enable debugging of source code. Through modification of the IL this association between IL and source code is lost. This is due to the fact that the connection made at compile-time is distorted in the IL modification. Run-time implementations also change the behaviour of an application and the connection between base and crosscutting source code cannot be made. This is because, within a run-time AOSD implementation, there is no connection made between concerns before run-time. Thus these approaches negate the possibility of supporting debugging without creating custom debuggers to support the approach being taken to support AOSD.

Approaches that modify languages through compiler alteration can support debugging of source code. This assumes that these compiler extensions are designed to create standard CLI compliant assemblies which can contain debug associations between source code and IL. These approaches, however, do not support cross-language weaving. AspectC# (Section 2.4.1), IlaC# (Section 2.4.2) and Aspect.Net (Section 2.4.3) are examples of this this type of approach.

The cross-language source code weaving approach taken by SourceWeave.net supports debugging of source code. When executed, SourceWeave.Net firstly parses source code to an AST, weaves the AST and then returns the woven AST back to the original language the AST was expressed in. This woven source code is then compiled. There is an association made during compilation between the woven source code and IL which enables the debugging of woven source code. Thus when a developer debugs an application made up of base and aspect components written in differing languages, the woven source code that the component was written in the same language it was originally expressed in before weaving.

Debugging is an important facility to many programmers developing software . This approach allows cross-language weaving of source code and debugging of that woven source code. In this way this approach represents major benefit to developers who are want to use AOSD concepts but need access to a source code debugging facility to debug woven systems.

5.2.6 Effects on Security

The .Net Framework is aims to provide a platform that enforces security at all levels of the .Net framework. This security focus can be seen in the CLS, as all languages are designed to support safe application development, but also in the CLR in which an evidence base code access security model has been implemented.

Implementing AOSD as a modification to the CLR can be also regarded as a possible security issue. As the .Net security model is built into the run-time environment then modification of the CLR may be seen

as a threat to the security model. Being able to alter an application's behaviour at run-time may serve as a way to modify the application in a manner that was not intended by the creator of the application.

The idea that a developer can modify the behaviour of an assembly though weaving additional behaviour to that assembly, in a way that was unintended by the creator of an the assembly, raises some security issues for IL level weaving as a potential migration for .Net to AOSD.

With SourceWeave.Net this is not a problem as the weaver works with source code. This is assumes that only parties who are permitted to modify a component through weaving, can obtain source code for the component. Under this assumption, SourceWeave.Net is does not impact on the security models that have been implemented within the .Net framework.

5.2.7 Developer Orientation

The .Net platform aims to support RAD, and any approach which implements the integration of the AOSD paradigm on .Net platform should support this aim.

IDE Integration

SourceWeave.Net is also integrated into Visual Studio.Net. Within this environment a developer can just point and click on a "weave" button and all the input detection, weaving, output creation, compilation and execution of the application is automated. SourceWeave.Net is the only approach that has been integrated into a development environment.

Approach Transparency

As discussed in section 5.2.5, after weaving the source code is output in the language it was originally expressed in. Thus a developer can look at this woven source code to ensure that code has been altered in the manner that the developer expects.

SourceWeave.Net has demonstrated, through the points made here and in section 5.2.5, that this approach is developer oriented. Developers need some degree of transparency to deal with situations where problems occur in weaving, but also when the technology is new as is the case with AOSD. Being able to see the before and after effects of weaving in the source code provides a developer a level of transparency as the developer can see the results of weaving in the source code.

IL based approaches do not provide the developer the transparency of a source code based approach. When weaving IL, the output of the weaver is IL. Unless the developer can read IL, which developers generally do not, this approach is not transparent to the developer. The only way the developer can assess the level of weaving that has been achieved is to observe how the behaviour of an application has changed post-weave. Run-time approaches suffer from the same problem. A developer cannot see what

is happening at the run-time level and as such there is no transparency in a run-time based approach. Again only through observing how behaviour is modified in the application as it executes, can a developer assess if the joinpoints that they intended to be crosscut with additional behaviour at run-time have been identified and crosscut.

5.3 Limitations of SourceWeave.Net

This work has highlighted some of the limitations associated with the approach taken in SourceWeave.Net.

In the following sections we detail those limitations.

5.3.1 CodeDOM Dependency

The set of language languages that SourceWeave.Net can support is constrained by the need for the language to implement the CodeDOM interfaces used within the implementation to compile and woven CodeDOM source code representations to assemblies and convert the woven CodeDOM source code representations back to the language the abstracted source code was originally expressed in.

Thus only languages that support CodeDOM can be implemented within SourceWeave.Net. Language providers are required to provide support for CodeDOM however it is really up to the language provider how much they want to implement.

This dependency on third party implementation of CodeDOM interfaces is a limitation of this approach.

5.3.2 Limitations of CodeDOM

CodeDOM is axial to this implementation of source level weaving. the abstraction it provides supports both joinpoint detection and weaving. However, CodeDOM does have significant limitations. This is possibly due to its narrow scope of intended usage. The CodeDOM namespace has been designed to support ASP.Net Assembly generation, WSDL proxy generation, Managed C++ WinForms designer base code generation and Visual Studio.Net code wizards.

CodeDOM also seems to be based on the C# language, in that it is much more natural to create CodeDOM representations of C# source code than representations of Eiffel source code. For example, CodeDOM does not support many constructs that are language specific such as VB modules and Eiffel design by contract constructs such as pre, post and invariant conditions. Mainstream languages such as C#, VB.Net, C++ and J# are in this way are more naturally mapped to CodeDOM that other languages such as Eiffel, Python and COBOL.

CodeDOM does not support many of the other common language constructs such nested namespaces, variable declaration lists, namespace aliasing, pointers, unary expressions and "jagged" or nested arrays.

CodeDOM does not support the safety modifiers such as readonly and volatile. Thus, the safety modifiers that a developer specifies in source code cannot be represented abstractly. Other structural safety modifiers such as virtual and override are not supported in events, operator members and destructors in the C# CodeDOM provider are not supported. Add and Remove accessors for events are not supported.

CodeDOM provides very little support for attributes. Attribute targets and accessors on attributes are not supported.

CodeDOM does not support the unsafe keyword used to denote unmanaged code.

Because of these CodeDOM limitations SourceWeave.Net cannot work with unmanaged code, support joinpoints in constructs that are not shared by all .Net languages, support for many commonly used programming techniques or ensure the behavioural and structural safety of woven source code.

5.3.3 Source Code Availability

SourceWeave.Net can only work in situations where source code is available. SourceWeave.Net cannot work with components for which source code is not available. Thus, components for which source code is not available cannot be used within SourceWeave.Net.

5.3.4 Compilation Dependencies

SourceWeave.Net is limited because of compilation dependency issues. For example, if there are three compilation units A, B and C, then if A references B, and a reference to C is woven into B a circular compilation dependency results. None of these compilation units of source code can compile as each compilation unit needs another to be compiled for itself to be compiled. This deadlock situation cannot be resolved by SourceWeave.Net and as such is a limitation of this approach.

This is not a problem for IL and run-time weaving as compilation has occurred and there is no compilation dependency issues.

5.4 Summary

In this chapter we presented an evaluation of SourceWeave.Net against other approaches aimed at introducing AOSD concepts to the .Net framework. This chapter also illustrates the limitations of this approach.

Chapter 6

Conclusions

6.1 Introduction

In Chapter 5 we presented an evaluation of SourceWeave.Net against other approaches aimed at introducing AOSD concepts to the .Net framework. Chapter 5 also presented the limitations of SourceWeave.Net. In this chapter we will draw conclusions from this evaluation to explore the suitability of the cross-language source code weaving approach implemented in SourceWeave.Net as a migration path for the introduction of AOSD concepts to .Net framework.

6.2 Conclusions

We believe that cross-language source code weaving, the approach implemented by SourceWeave.Net, has strong potential as a migration path for the introduction of AOSD concepts to the .Net framework. In this section we will discuss the benefits and drawbacks of this approach in relation to the other approaches to implementing AOSD within the .Net platform. Through this discussion we hope to illustrate why we consider cross-language source code weaving to be a possible migration path for the introduction of the AOSD paradigm to .the Net framework .

Cross-language source level weaving, as discussed in Section 5.2.1, has been demonstrated through SourceWeave.Net as being able to support an extensive and extensible cross-language joinpoint model. Although similar IL based approaches also support extensive joinpoint models they do not support the scope for extensibility that source code weaving provides. One limitation, in terms of the joinpoint support available in the SourceWeave.Net, lies in the limitations of CodeDOM. CodeDOM does not support certain language constructs and thus joinpoints to represent these constructs cannot be supported. Another limitation is based on compilation dependencies, which is discussed in Section 5.3.4. This is when woven code will not compile because of circular dependencies that are formed during source weaving.

These limitations however do not seriously detract from the joinpoint model implemented and value to a developer.

As discussed in Section 5.2.3, run-time and IL based weaving approaches are language-independent and as such implicitly support cross-language weaving. The J# language is the exception to the rule and cannot be utilised in IL based weaving approaches, such as Weave.Net. SourceWeave.Net currently supports J#, C# and VB.Net, but can support any .Net language that implements CodeDOM tools. This dependency of cross-language source weaving on CodeDOM, illustrated in Section 5.3.1, is however a constraining factor of this approach.

Although modification of the CLR supports the full range of .Net languages, the alteration of the .Net framework to enable weaving is a major undertaking which also affects run-time performance. Changing the framework to integrate AOSD concepts, as presented in Section 5.2.4, means altering an architecture that has been specified and designed to integrate may facilities and features which promote RAD in a multi-language environment. Modification of the framework support AOSD could result a loss of these facilities.

Debugging, as demonstrated in Section 5.2.5, is example of a facility that is lost through implementation of AOSD concepts at both run-time and pre-run-time via IL weaving. Security models are also threatened by these approaches; this point is illustrated in Section 5.2.6. The .Net framework security models have not been designed with integration of AOSD concepts in mind. Thus, introduction of AOSD concepts in the lower layers of the .Net framework is dangerous in terms of security and safety models that could be affected.

The cross-language source code weaving approach implemented in SourceWeave.Net avoids all of these problems by not altering any part of the .Net framework beyond the base input, which is source code. Thus there is no impact on any facilities that a programmer uses within the framework. This approach protects the investments made in terms of time and money creating the .Net framework. A limitation of working with source code is that SourceWeave.Net must have source code to work, so a developer cannot weave behaviour into a component that there is no source code for.

In summary, SourceWeave.Net's implementation of cross-language source code weaving supports an extensive and extensible joinpoint model, can potentially support the full spectrum of .Net languages and does not incur any of the problems associated with framework alteration. Although we believe this approach has promise this is a proof-of-concept and much work into the further investigation of cross-language source code weaving needs to be undertaken for this approach to be seriously considered as a full migration option.

6.3 Future work

This section details areas of future work that are related to this dissertation.

6.3.1 CFlow

Cflow joinpoints have not been implemented in the SourceWeave.Net joinpoint model. An extension of the joinpoint model to include CFlow joinpoints is the next step in for SourceWeave.Net to realise the full joinpoint model that has been bench-marked in the AspectJ joinpoint model. This extension would bring this approach closer to being a full migration possibility.

6.3.2 Aspectual Polymorphism

Aspectual polymorphism has not been fully implemented by any other AOSD tool on the .Net framework. SourceWeave.Net could be extended to introduce aspectual polymorphism.

6.3.3 Extension of Language base

SourceWeave.Net has been designed to support the full compliment of .Net compliant languages. At present, SourceWeave.Net supports three languages VB.Net, C# and J#. This language base supported by SourceWeave.Net should be extended to include all the .Net languages.

6.3.4 Language Based Joinpoint Model Extensions

There are certain constructs that are language specific which are points where crosscutting behaviour can exist, but in the current joinpoint model this behaviour cannot be separated and modularised. The joinpoint model of SourceWeave.Net could be extended to introduce certain language specific joinpoints. For example, we could introduce a pre or post condition joinpoint to allow modularisation of crosscutting behaviour of behaviour in the Eiffel language .

6.3.5 Instance Based Weaving

IlaC# discussed in Section 2.4.2 is attempting to implement instance level weaving within the C# language. This work could be extended by in SourceWeave.Net to create language-independent, source code based, instance level weaving.

6.3.6 Genericity

Genericity will to soon be introduced to the .Net framework. This will have consequences for the source code weaving as new constructs may be introduced into languages to support genericity. Also genericity may provide a basis for extension of the joinpoint model. Research into how the joinpoint model will be affected through the introduction of genericity and how the joinpoint model can be augmented to allow

further modularisation of crosscutting concerns through the utilisation of generics is a an area needs to be investigated.

6.3.7 Extending CodeDOM

This work has uncovered some major limitations of the CodeDOM model in abstractly representing all source code constructs. Research into how these limitations might be overcome would be of great help this work to others who wish to continue the research into cross-language source code weaving.

6.3.8 Source/IL Hybrid Weaver

Many of the limitations associated with source code weaving are resolved in IL weavers and vice-versa. Research into the possibility of creating a hybrid weaver that can work with both artifact types may solve some of the limitations faced by either approach. As such this warrants further investigation.

6.3.9 CME

The CME environment discussed in Section 2.5, is a new AOSD tool development platform currently being developed by IBM. The CME is to provide standardised basis on which language-independent AOSD approaches can be developed. This is a new and interesting concept in that it aims to provide many of the support that is needed to implement any AOSD approach. Porting SourceWeave.Net to this new environment might serve as a beginning towards a level of standardisation in the creation to AOSD tools.

6.3.10 Multi-Dimensional Separation of Concerns

SourceWeave.Net currently emulates the AspectJ model, in that it provides an extra dimension to separate concerns in the single dimensional OO paradigm. The AspectJ community as discussed in section 2.3.2, has begun to look at how the number of dimensions through which concerns can be separated can be increased. In this way AspectJ is moving towards the multi-dimensional separation of concerns.

SourceWeave.Net could be extended in the same way as the model of SourceWeave.Net is similar to that of AspectJ. No other approach to AOSD introduction on the .Net platform has implemented multi-dimensional separation of concerns.

6.4 Summary

This chapter concludes this thesis. In this chapter we present the benefits and limitations of this approach and conclude that this approach is a strong migration path for the introduction of AOSD to the .Net

framework. We then present the work that we believe is needed to fully implement this proof-of-concept implementation of cross-language source code weaving.

Bibliography

[1] John Lamping Anurag Mendhekar, Gregor Kiczales. Rg: A case-study for aspect-oriented programming.

[2] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters, 2001.

[3] Simon Fell Dharma Shukla and Chris Sells. Aspect-oriented programming enables better code encapsulation and reuse - http://msdn.microsoft.com/msdnmag/issues/02/03/aop/default.aspx, 2003.

[4] Cormac Driver. Evaluation of aspect-oriented software development for distributed systems, 2003.

[5] Eclipse. Eclipse - http://eclipse.org/aspectj/, 2003.

[6] Jim Farley. Microsoft .net vs. j2ee: How do they stack up?- java.oreilly.com, 2000.

[7] Robert J. Walker Gail C. Murphy, Albert Lai and Martin P. Robillard. Separating features in source code: An exploratory study. In *International Conference on Software Engineering*, pages 275–284, 2001.

[8] Anurag Menhdhekar Chris Maeda Cristina Lopes Jean-Marc Loingtier Gregor Kiczales, John Lamping and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[9] Jim Hugunin Mik Kersten Jeffrey Palm Gregor Kiczales, Erik Hilsdale and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[10] ivan zderadicka. Csharp to codedom parser - http://ivanz.webpark.cz/csparser.html, 2003.

[11] Howard Kim. Aspectcsharp: An aosd implementation for csharp., 2002.

[12] Donal Lafferty. W3c xml schema for aspectj aspects survey of aspectj grammar., 2003.

[13] Donal Lafferty. Weave.net - http://www.dsg.cs.tcd.ie/, 2003.

[14] John Lam. Claw - cross language aspect weaving, demonstration aosd 2002, enschede, 2002.

[15] Microsoft. Rotor project sscli - http://research.microsoft.com/collaboration/university/europe/rfp/rotor/default.aspx, 2003.

[16] Ximian mono. http://www.go-mono.org, 2003.

[17] MSDN. System.codedom.compiler - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcodedomcompiler.asp, 2003.

[18] msdn - EMCA. EMCA - http://msdn.microsoft.com/net/ecma/. , 2003.

[19] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, 1999.

[20] Hridesh Rajan and Kevin Sullivan. Need for instance level aspect language with rich pointcut language., 2002.

[21] Prof. Vladimir O. Safonov. Aspect.net a framework for aspect oriented programming for .net platform and csharp language - msr rotor workshop cambridge, 2002.

[22] Fabian Schmied. Aop.net - http://wwwse.fhs-hagenberg.ac.at, 2003.

[23] Wolfgang Schult and Andreas Polze. Aspect-oriented programmingwith csharp and .net., 2002.

[24] MSDN System.CodeDom. System.codedom - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcodedom.asp, 2003.

[25] Watson Tarr and Harison. Concern manipulation environmemt - http://www.research.ibm.com/cme/vision/vision.htm, 2002.

[26] Robert Tolksdorf. Programming languages for the java virtual machine, 2003.

[27] Peter Tröger, Wolfgang Schult, and Andreas Polze. Loom.net - http://www.dcl.hpi.uni-potsdam.de/cms/research/loom/, 2003.

[28] J.C. Wichman. Composej: The development of a preprocessor to faciliatate composition filters in the java language, 1999.

[29] Wrox. Professional .net framework - http://www.wrox.com/, 2001.