AspectC#: An AOSD implementation for C#.

Howard Kim
Department Of Computer Science
Trinity College Dublin

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

September 13, 2002

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: ______________

Date: September 13, 2002

Permission to lend and/or copy

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

Signed: ______________

Date: September 13, 2002

Acknowledgments

I would firstly and most importantly like to thank Dr. Siobhn Clarke for introducing me into the world of AOSD and for her help, advice and guidance both inside and outside office hours throughout the year. Her faultless guidance allowed me to be lucky enough to have a paper published at a workshop at the first international conference on AOSD in Utrecht in April 2002. Within the pages of this dissertation lie the bloom of the seeds Dr. Clarke laid throughout my postgraduate year studying under her, any errors, omissions or mistakes are mine and should not reflect on her. I hope the quality of this dissertation is a small reflection on the hard work put into it by Dr Clarke, thank you!

I'd like to thank: All the members of lecturers/staff at the Computer Science Department for their help throughout all my years at TCD. To the lecturers of the NDS course for their help and advice throughout the year. The students of the NDS class 2001-2002! Cormac for his help with AspectJ, Hyper/J and all things with aspects. Martin for sitting beside me all year! Rich Greenane for his help with Texnic and images. Simon Dobson for his advice and help with latex. Owen for his help with the thesis. To all my friends for their support.

Finally but certainly not least, I'd like to thank my family. To my mother for all her love and support. To my brothers Jong and Oisin for their proof reading, suggestions, advice and support. Thank you!

Abstract

A major problem with Object Oriented Programming (OOP) is that it cannot deal efficiently with crosscutting concerns. Aspect Oriented Software Development (AOSD) is a new methodology that tries to enable the extension of the separation of concerns capabilities in software development. AOSD, which encompasses Aspect Oriented Programming (AOP), hopes to enable the developer or architect to capture crosscutting concerns in a modular fashion.

The objectives of the dissertation include the design, implementation and evaluation of a tool that enables the modularisation of crosscutting concerns within C#, that we have called "AspectC#". AspectC# must:

- Allow the developer to use AOP constructs within C#.
- Make no explicit language extensions to C#.
- Be extensible for future development.

AspectC# hopes to enable the developer to modularise crosscutting concerns within C#. By modularising crosscutting concerns this will lead to an easier development and maintenance of applications.

The evaluation included a small case study comparing an example using AspectC# and OOP with C#. We believe the case study and tool supports the case for the use and promotion of AOSD. Special emphasis was placed on picking a real world, easily understood example of the use of the tool and we believe this represents an excellent introduction to some of the most widely used AOP and AOSD techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Since the invention of the computer programming languages have evolved from basic machine code and assembly languages to procedural and object oriented (OO) languages. Each progression in technology has enhanced the ability to make a clean separation of concerns in the source code, which in turn allowed easier development and maintenance. For the past decade the predominant way of programming has been using Object Oriented Programming (OOP); OOP can deal effectively with real world problems because the object model can encapsulate real world objects.

1.2 Addressing the Problem

Separation of concerns is at the heart of software engineering and has been attributed to Dijkstra & Parnas [Dij76] [Par72]. Ossher [OT01] describes separation of concerns by saying: "*In its most general form, it refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal or purpose.*". Therefore a '*concern*' can be thought of as module that encapsulates a particular area of interest. Concerns are the main reason for decomposing software into smaller parts or elements. There exist many types of concerns that are relevant to developers at different stages in the development life-cycle. With regard to OOP , a *class* is the encapsulation of a concern within the OO paradigm.

As stated, the main concept of OO is that everything is modelled within a class structure; this technique is effective for some real world situations as the object model provides a general fit to these problem domains. OOP has many advantages but there are also many limitations to OOP. Many problems occur with OO techniques when concerns *crosscut* each other. When we state that concerns crosscut we mean that some concerns are localised within a single system element (e.g. class or method) and other span or cross multiple system elements. They 'cross cut' multiple system elements. This leads to problems identified by Kiczales [KLM$^+$97], which are:

1. Code Tangling: Occurs when multiple concerns are implemented within the same system element. This has the affect that the elements are more complex to understand and maintenance becomes more difficult.

2. Code Scattering: Occurs when a concern results in the implementation of duplicate code or different code that is distributed across multiple system elements. This makes maintenance harder because changes in the code must be replicated across all elements, therefore the developer must understand all elements within the system. This results in greater chance of inconsistencies and mistakes.

1.3 Motivation

In a paper by Kiczales et al. [KLM$^+$97] the term *'aspects'* was coined. An aspect is a modular unit of a crosscutting concern. The programming technique that addressed these design issues was called Aspect Oriented Programming (AOP).

Aspect-Oriented Software Development (AOSD) is a new technology that tries to enable the extension of the separation of concerns capabilities in software development. There has been much research into AOSD with the key goal being better separation of concerns. Aspects may appear at any stage within the software development lifecycle and common examples of aspects or concerns are: security, quality of service (QoS), transactions and synchronisation.

AOSD, which encompasses AOP, hopes to enable the designer or architect or developer to capture crosscutting concerns at any stage within the development lifecycle, this will lead to a more modularised design and implementation and also easier maintenance.

1.4 Goals

The objectives of the dissertation include the development a tool that we have called *AspectC#*, which enables the modular organisation of crosscutting concerns within the C# language. This should also include an investigation and evaluation of the .NET platform as an AOSD platform. In creation of this tool a thorough evaluation of both the .NET Framework and currently available AOP tools is necessary. The dissertation goals are:

1. To design and implement a tool (AspectC#) that enables the developer to use AOP mechanisms within C#.

2. Evaluation of AspectC#.

3. Investigate and evaluate the .NET Framework with regard to the potential of AOSD within the Framework.

4. Investigate current AOP technologies.

If we elaborate on the design and implementation of AspectC# the specific goals with regard to AspectC# are:

- Enable the developer to use AOP mechanisms within C#: This goal will enable developers of C# applications to modularise crosscutting concerns by using our tool.
- Make no language extensions to C#: By not extending the C# language there is no need to develop a compiler for AspectC#, it also has the effect that developers can write applications within Visual Studio .NET environment.
- Be extensible for future development: AspectC# is currently limited to just C#, we hope that in future versions other .NET supported languages may be used for cross language aspect weaving. It is also a goal that the semantics of AspectC# may be extended.

1.5 Thesis Outline

Chapter 1 *Introduction* attempts to introduce the reader to some of the background on programming languages, the motivation for the project, what are the goals of the project and what is the aims of AspectC#.

Chapter 2 *State of the Art* reviews the state of the art of Aspect Oriented Software Development. This includes Java and .NET based tools that provide AOP mechanisms.

Chapter 3 *Requirements and Design* gives a detailed Requirements Specification and gives an overview of how components of the AspectC# tool interact with each other.

Chapter 4 *Implementation* describes the implementation of AspectC#. It details how particular design features were implemented in the system. Small simple code examples are described to reinforce points.

Chapter 5 *Evaluation* attempts to provide an analysis of the currently available tools for AOP. Also an evaluation of AspectC# is performed using a small case study.

Chapter 6 *Conclusion* reviews the goals and how they were met, it then suggests areas of AspectC# that warrant future work and research.

1.6 Summary

In this chapter a basic introduction into reasons for AOSD has been discussed; how the object model does not fit all problems and how this leads to the problems of code tangling and scattering. AOSD technology is currently in a nascent stage but there are already a large number of research groups investigating the possible future directions of AOSD.

Chapter 2

State of the Art

2.1 Introduction

In this chapter, we examine the state of the art of the .NET Framework and look at the latest AOP technologies for both Java and .NET languages. The chapter begins with an assessment of the .NET Framework and its encompassing tools and technologies. Next there is an overview of AOSD. Finally we examine the latest approaches in AOP tools for both Java and .NET languages.

2.2 .NET Framework

The .NET initiative began in 1997 when Microsoft decided that it would be the successor to Windows$^{\text{TM}}$. The .NET Framework is the base for the .NET initiative. In the most general sense the framework consists of a set of types, classes, services and tools that are combined to form the new platform. The framework has been designed to address the limitations of Windows, which included complex development and complex deployment. The goals of the .NET Framework are:

1. New development platform: provide a development platform for both Internet and distributed applications.

2. Simplify development of applications: Enable easy development and deployment with better tools.

3. Improve interoperability and integration: Easier integration between system and applications.

4. Support for any device *“Universal access”*.

2.2.1 Architecture

The .Net Framework is composed from several layers that provide services, tools and classes for the development platform. Figure 2.1 shows a block diagram of the .NET framework. The framework consists of:

- Languages and developer tools: Visual C# .NET (C#), Visual Basic .Net (VB.NET) , Visual C++, Visual J# and other 3rd party languages.

- Base Class library: The base class library is a collection of reusable types that tightly integrate with the common language runtime. The base class library forms the .NET framework Software Development Kit (SDK), which provides access to system functionality and is designed to be the base upon which .NET components, applications and controls are built.

- Common Language Runtime (CLR): provides a reliable, secure execution environment and also multiple language support.

2.2.2 Languages

The languages currently supported by the .NET framework are: C#, VB.NET, C++, J#, Jscript, Eiffel.NET, Cobol, Small Talk, Perl, Pascal, Python, Oberon, Haskell, Mercury, Scheme, CAML, Component Pascal, Mondrian, Fortran, Mercury, Dyalog APL, CLisp and OZ.

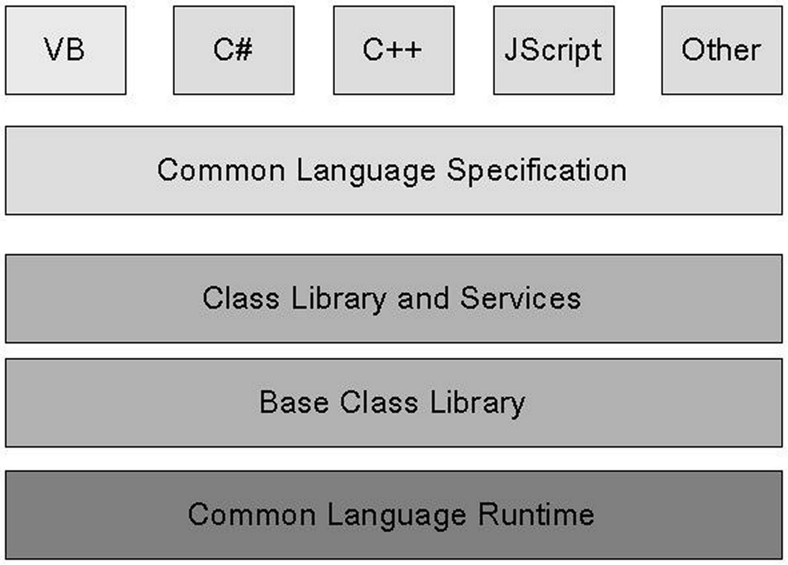

Figure 2.1: .Net Framework Block Diagram.

2.2.3 Base Class Library

The .NET framework provides a large and extensible class library. The class library framework is organised into a hierarchical structure that can be used across any of the supported .NET languages.

Of special interest to the project is the Code Document Object Model (CodeDOM) namespace which is part of the base class library. The CodeDOM namespace provides interfaces, classes and architecture to represent the structure of a source code document independent of language as an object graph or tree. The CodeDOM API also provides a set of classes for compilation of the object graph into a supported language. Figure 2.2 shows the hierarchical structure of the CodeDOM API. Also, we can see in figure 2.3 where exactly CodeDOM is positioned within the .NET Framework.

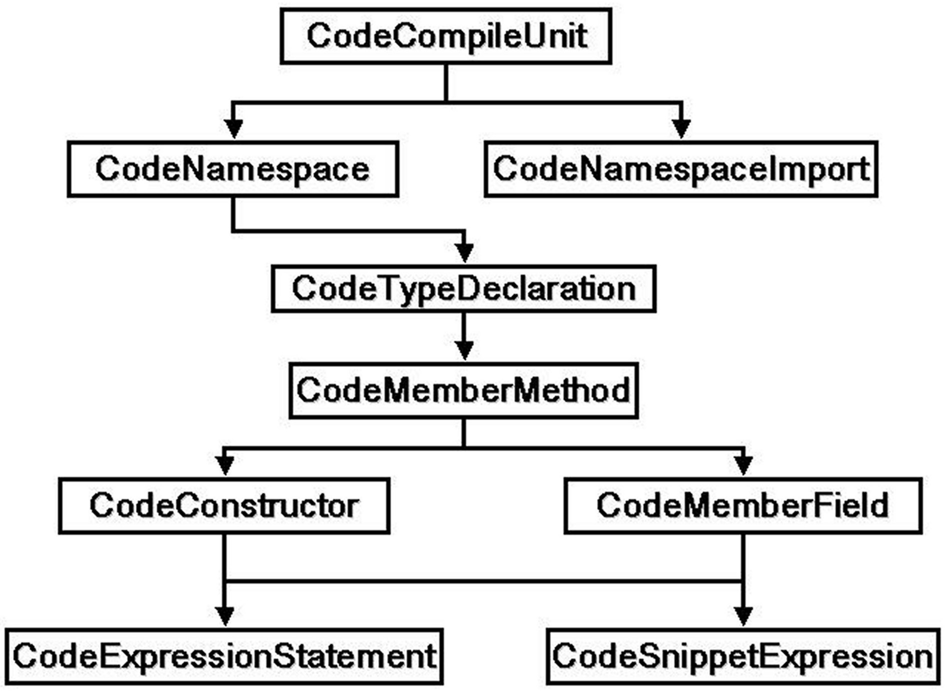

Figure 2.2: CodeDOM Hierarchy

Hello World example using CodeDOM

This example shows how to create a C# class called HelloWorld that prints to the console HelloWorld using CodeDOM. The code examples are written in C# using the CodeDOM API. From figure 2.2 we can see that firstly we need to create a CodeCompileUnit [1], then added to the compile unit is the namespace followed by any import statements. Next is the actual class and finally the Main method is added.

[1]The CodeCompileUnit is the only object that can be directly compiled using the CodeDOM compiler. The CodeCompileUnit represents the main reference point of the object graph and elements such as namespaces are added to the compile unit.

Compile Unit The compile unit references a CodeDOM tree that can be compiled. CodeDOM compilers process the tree referenced by the compile unit. The `CodeCompileUnit` class defines the compile unit and source files already in the target language can be defined by using the `CodeSnippetCompil-eUnit` class. The following code snippet creates a new compile unit.

```
CodeCompileUnit compileUnit = new CodeCompileUnit();
```

Defining and importing a namespace Defining a `namespace` with CodeDOM is done by creating a `CodeNamespace` object. To add namespace imports to the namespace we can call `CodeNamespace.Imports.Add` method.

```
CodeNamespace myNamespace = new
            CodeNamespace("HelloWorld");
myNamespace.Imports.Add("System");
```

Defining a class To declare an interface, class, struct or enumeration in CodeDOM we use `CodeTypeDeclaration` class. As shown below:

```
CodeTypeDeclaration myClass = new
               CodeTypeDeclaration("HelloWorld");
MyNamespace.Types.Add(myClass);
```

Adding class members to a class The CodeDOM model provides a variety of elements that can be added to a class. A new member can be added to a class using `CodeTypeDeclaration.Members.Add` method, as shown below:

```
CodeEntryPointMethod Main = new CodeEntryPointMethod();
CodeMethodInvokeExpression expression = new
   CodeMethodInvokeExpression (
       new CodeTypeReferenceExpression("System.Console"),
      "WriteLine",
       new CodePrimitiveExpression("Hello World!")
       );

myClass.Members.Add(expression);
```

2.2.4 Common Language Runtime (CLR)

The CLR provides a runtime execution environment for .NET applications; it is analogous to the Java Runtime Environment (JRE) from the Java Language ^{TM}. Code that is run under the control of the CLR is called *managed code*. The CLR manages the code i.e. compilation of the code into native code and memory management. Code that runs outside the control of the CLR is called *unmanaged code*, but it is possible to integrate unmanaged code within the CLR. The CLR can be divided into three main categories:

- The Type system.
- The Execution system.
- The Metadata system.

The Type system consists of the Common Type System (CTS) and the Common Language Specification (CLS). Types in the .NET framework can be either **value types** or **reference types**. Value types can either be built-in value types such as **int** or user defined value types. Reference types can be either user defined object, interface or pointer types. Figure 2.4 shows the hierarchy for the type system within the .NET framework. The CTS defines the standard value and reference types supported by the framework.

The CLS defines rules that a language must obey if it is to execute under the CLR. The CLS defines a subset of the CTS, that is the same rules apply to the CTS as to the CLS. The CLS also establishes requirements for compliance, these help to determine if the language conforms to the CLS. If a language conforms to the

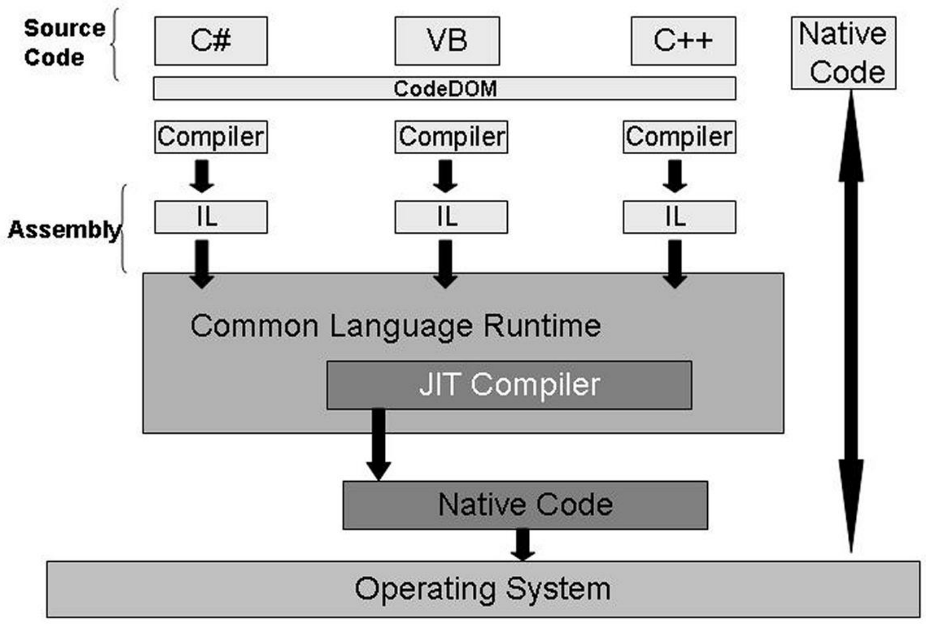

Figure 2.3: Execution Model .Net Framework.

CLS it may be used within the CLR and also by other .NET languages.

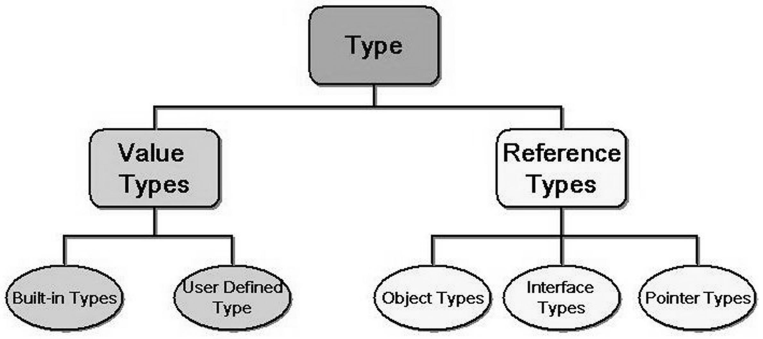

Figure 2.4: .Net Type System.

The Execution system is at the heart of the .NET framework. Source code written in any of the supported .NET languages is compiled into Microsoft Intermediate Language (MSIL). When the CLR executes an application the MSIL is 'Just In Time' (JIT) compiled into native machine code. It is the MSIL that allows for cross language integration of applications.

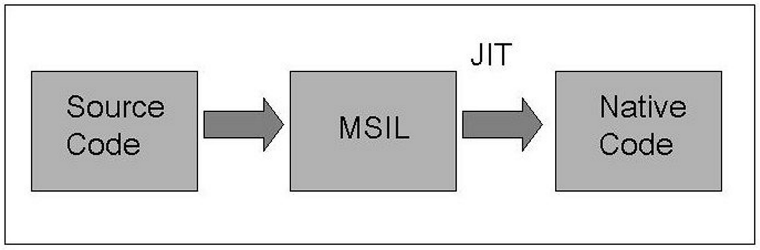

Figure 2.5: .Net Compiling

The Metadata system is an integral part of the framework. An assembly is a primary building block of a .NET application. In previous versions of Windows an application or program was usually installed using a Dynamic Link Library (dll) in .NET a managed dll is called an assembly. It is analogous to the Java Archive (jar) file within the Java Language. An assembly contains:

- Manifest - Describes the assembly, e.g. assembly name, version etc.
- Meta data about types - Type information about the classes is kept within the assembly.
- Modules - The module contains the MSIL code, this code is used by the CLR and JIT compiled before execution.
- Resources - Resources can be anything from text files to images, this enables the assembly to be self contained.

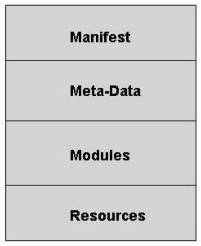

Figure 2.6: Assembly Structure

Assemblies are fundamental part of the development in .NET they perform functions such as:

- Contains code needed for execution within the CLR
- Forms the security boundary, permission are granted or revoked at this level.
- Type boundary, every type as an associated assembly.
- Reference boundary, the manifest of an assembly contains metadata used for resolving conflicts.
- Version boundary, an assembly forms the smallest version-able unit within the CLR.
- An assembly is the deployment unit within the CLR.

The .NET framework is an improvement on the Windows platform and addresses many of the limitations of Windows. But it more than just Windows++, the .NET initiative enables simplified development and deployment of applications within the Framework, with greater interoperability and integration and support for any device.

2.3 AOSD Technology

Aspect-Oriented Software Development (AOSD) is a new technology that extends the separation of concerns (SoC) in software development. The methods of AOSD facilitate the modularisation of crosscutting concerns within a system.

Aspects may appear in any stage of the software development lifecycle (e.g. requirements, specification, design, implementation, etc.). Crosscutting concerns can range from high-level notions of security to low-level notions like caching and from functional requirements such as business rules to non-functional requirements like transactions.

As described on the AOSD website [AOS] the main thrust of research in AOSD is into the extension of separation of concerns (SOC) techniques, but not only crosscutting techniques: “Researchers in AOSD are largely driven by the fundamental goal of better separation of concerns. So while crosscutting tends to be a significant focus of their work, they also incorporate other kinds of SOC techniques, including such well established approaches as OO and good old-fashioned structured programming. This is reflected in the way much of the work in the field blends support for many different kinds of modularity including block structure, object structure, inheritance as well as crosscutting.”.

A principal design model language in AOSD is Theme/UML [Cla]. Theme /UML presents a method for designing systems based on the object-oriented model but with added extensions or decomposition mechanisms. Each model consists of a *theme*, which is an individual requirement; extensions to Unified Modelling Language (UML)[Boo98] are required for the composition of the thematic design models. Theme/UML contains support for overlapping specifications and crosscutting specifications.

2.4 Java AOP Approaches

In this section we cover the principal Java approaches to AOP. Currently the principle approaches are AspectJ [Tea], Hyper/J [HYP], Composition Filters [COM] and DemeterJ [Lie]. AspectJ and Hyper/J will be covered in detail over the next two subsections.

2.4.1 AspectJ

AspectJ is a simple and practical aspect-oriented extension to Java that enables the modular implementation of crosscutting concerns [KHH+01]. It is also the basis for an empirical assessment of aspect-oriented programming. The designers of AspectJ would like to gain a better understanding of how AOSD and an AOP language can be use in a "real user community". The designers of AspectJ call a well modularised crosscutting concern an '*aspect*'.

The Language

AspectJ extends the Java Language^{TM} with support for two types of crosscutting implementation. The first type enables the developer to define additional implementation to run at certain well-defined points of execution in the program; this is called "*dynamic crosscutting*". The second type enables the developer to define new operations on existing types; this is called "*static crosscutting*".

Dynamic Crosscutting The dynamic crosscutting in AspectJ is based on set constructs. A *Join Point* is a well-defined point of execution in the program. *Pointcuts* are a mechanism of referring to a set of join points and certain values at those join points. *Advice* is a method like construct that enables additional implementation to be inserted at join points. An *aspect* is a modular unit of a crosscutting concern; composed of pointcuts, advice and Java declarations.

Static Crosscutting The static crosscutting in AspectJ is achieved by *introduction*. New member variables and methods can be introduced statically from aspect classes into base classes. This is a powerful construct as not only does it change the behaviour of a class it can change the relationship between classes.

Join Point Model

The Join Point model is a critical element in the design of any AOP language. In AspectJ the join point model is defined as well-defined points of execution in the program. In this model join points can be considered to be where method calls or object field accesses are executed, or on exceptional handling or static initialisation within a class or method. Table 2.1 shows the full list of the dynamic join points in AspectJ.

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

Table 2.1: Join Point Model AspectJ

Pointcut Designators

A *pointcut* is a set of join points. AspectJ provides several pointcut designators for associating advice with join points. For example the following code matches any call to method `setX()` of an object of type Point. Pointcuts may also be combined using ||, && and ! operators.

```
receptions(void Point.setX());
```

AspectJ allows for advice to have access to certain values at join points; in advice declarations values can be passed from the pointcut designator to the advice. In the next example the variables `p` and `nval` are available for use in the advice body.

```
before(Point p, int nval) : receptions(void
       p.setX(nval)) {
       System.out.println("x value of " + p +
                      " will be set to " + nval);
}
```

Advice

Advice is a method like mechanism used to declare that certain code should be executed at join points in a pointcut [KHH+01]. AspectJ supports three types of advice (before, after and around). Table 2.2 explains each type of advice.

Advice Type	Description
before	Advice executed before 'base' method at join point.
after	Advice executed after 'base' method at join point.
around	Advice selectively pre-empts normal execution.

Table 2.2: Advice in AspectJ

Advice declarations define advice by associating the type of advice with a join point (when it should be executed) and the code associated with the advice with the base code of the method. For example the following advice is associated the method `move()` and it is of type *after*; it associates the code `flag = true;` with the method:

```
after(): moves() {
   flag = true;
}
```

Around advice dynamically alters the behaviour of a method by running instead of the method. The original method can be called by using the `proceed` keyword. In the next example the original method is called by using `proceed` keyword but only positive values are passed to the method as `Math.max(int, int)` returns the greater of the two numbers.

```
around(int nv) returns void:
   receptions(void Point.setX(nv)) ||
   receptions (void Point.setY(nv))
   {
      proceed(Math.max(0, nv));
   }
```

Aspects

Aspects are modular units of crosscutting implementation [KHH+01]. An aspect is defined by aspect declarations, which have a similar form to the class structure. An aspect can consist of:

- Pointcut declarations
- Advice declarations
- Normal class declarations

The following code segment is an example of a tracing aspect. It associates advice with methods from the classes `FigureElement`, `Line` and `Point`.

```
aspect MoveTracing {
   static boolean flag = false;

   static boolean testAndClear() {
      boolean result = flag;
      flag = false;
      return result;
   }

   pointcut moves() :
   receptions(void FigureElement.incrXY(int, int) ||
   receptions(void Line.setP1(Point) ||
   receptions(void Point.setX(int) );

   after(): moves() {
      flag = true;
   }
}
```

Aspect precedence refers to the fact that more than one piece of advice may apply to a join point. There exists certain rules that determine which advice should run depending on if an aspect inherits from the other, or if the `dominates` modifier is used.

Reflective Access to Join Point

AspectJ provides access to the current join point by means of a special variable called the `thisJoinPoint`. It is bound to the current object and represents the join point.

Property Based crosscutting

It is possible within AspectJ to use wildcarding of pointcut designators. This can be simple pattern matching like `receptions (* Point.*(..))`, which matches any method call or execution on a Point object. More advanced pointcut designators are allowed using the `cflow` keyword to select pointcuts that are determined based on whether the join points are in a particular control-flow relationship with other join points.

Implementation

The main work of any AOP language is to ensure the correct combination of aspect and non-aspect code. This coordination process is called *aspect weaving* [KHH$^+$01]. In AspectJ the weaving is best described as a compile time weaver; the compiler transforms the source program in three ways:

- Advice declarations is compiled into standard methods.
- Parts of the program where advice is applicable are transformed corresponding to the join points
- Any parts of the program with dynamic dispatch code is inserted at those static points.

Relevance to AspectC#

AspectJ makes simple language extensions to Java that enables Java to modularise crosscutting concerns. There are many features of AspectJ that will be useful in AspectC#, the Join Point model is very effective and captures crosscutting concerns well. The different types of crosscutting identified by AspectJ will also be used within AspectC# (I.e. Static and dynamic).

2.4.2 Hyper/J

In Tarr et al [TOHS99], they describe how modern languages and methodologies suffer from a problem they call *"tyranny of dominant decomposition"*. Examples of tyranny are classes in OO and functions in Functional languages. This problem leads to the developer having to commit to one of these dominant decompositions early in the design and changes to this decision can effect the whole application. Often the developer is not given a choice about the decomposition dimension e.g. in OO it must be a class. The multi-dimension separation of concerns breaks the tyranny of dominant decomposition by not relying on a single decomposition mechanism. Hyper/J is an evolving tool that provides hyperspace support. Hyper/J ^{TM} supports "Advanced Multi-Dimensional Separation of Concerns(MDSOC) in standard Java ^{TM} software" [TO00]. Hyper/J provides a powerful composition model where separated concerns can be integrated into a program or component. Hyper/J can be used at any stage in the software development lifecycle: design, implementation, or maintenance.

The Language

The approach taken by the Hyper/J team is called the '*hyperspaces approach*'. Hyperspaces permit the identification and encapsulation of any arbitrary concern at any stage with the development lifecycle. It is also possible to manage the relationship between concerns. Figure 2.7 highlights the different terms within Hyper/J.

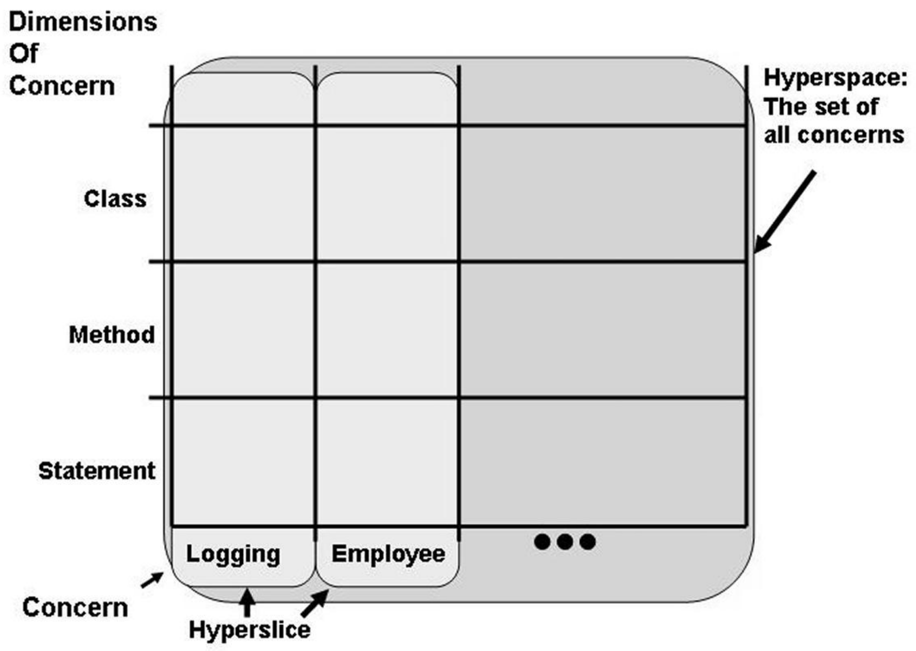

Figure 2.7: Hyper/J overview

Concern Space of Units

A unit can be thought of as construct in a language; it may be, for example a statement, state chart, class, interface or any entity within a given language. There exists two types of units: *primitive* and *compound* units. Primitive units can be treated as atomic, while compound units are grouped together. A *concern space* encompasses all units in a piece of software. The purpose of the concern space is to group all units in the system so that it is possible to separate all concerns and to describe the inter-relationship between those concerns.

- Identification: refers to the process of selecting concerns and populating them with units that are relevant to them.
- Encapsulation: refers to how after identification concerns are encapsulated. E.g. method, class or package.
- Integration: Once concerns have been identified and encapsulated they must then be integrated so the system addresses multiple concerns.

Identification: Concern Matrix

A *hyperspace* is a concern space specially constructed to support multi dimensional separation of concerns. Units are organised into a multi-dimensional matrix where each axis represents a dimension of concern and each point on an axis is a concern in that dimension. Each dimension can be viewed as a section of the set of all units.

A hyperspace file specifies the Java classes that contain code units that will populate the hyperspace. The following code segment shows a piece of a specification file. The specification file can contain either Java fully qualified class names or path name of files.

```
hyperspace Expression_SEE_Hyperspace
   class com.ibm.hyperJ.ExpressionSEE.*;
   file c:\u\smith\com\ibm\hyperJ\util\Set.class
```

Units Hyperspaces can organise and manipulate units in any language but currently only Java is supported. In Java, member variables and functions are treated as primitive units and classes, interfaces and packages are treated as compound units.

Concern Specifications The concern specification serves to identify the dimension and concern in the concern matrix. A simple approach is to use a set of *concern mappings*, such as:

```
x: dimension.concern
e.g.

package com.ibm.hyperJ.ExpressionSSE: Feature.Kernel
operation display :                    : Feature.Display
operation check :                      : Feature.Check
```

Where x if the name of the unit.

Hyperslices

The concern matrix identifies and manages units within a given dimension and concern. To support the encapsulation of concerns Hyper/J uses an additional mechanism called a *hyperslice*; a hyperslice is a set of concerns that are *declaratively complete* [2].

The declarative completeness means that there is no coupling between hyperslices. Instead of one hyperslice referring to another, an abstract declaration of what the hyperslice needs is declared. This has the effect that the hyperslices are self-contained, but of course someone will have to implement the abstract declarations.

[2]Declaratively complete means that anything a concern uses must be declared within the concern.

Relationship between concerns

Units, concerns and hyperslices may be interrelated in a number of ways. Hyper/J has two classifications of relationships:

- *Context-insensitive* were concerns are always related the same way, as long as they share units in common.
- *Context-sensitive* relationships depend on the context that the concerns are being used for.

Hypermodules

Hyperslices are the building blocks for system in Hyper/J. The binding relationship between units is called *correspondence*; correspondence is a context-sensitive relationship and occurs in the integration of the component or system. This integration context is called a *hypermodule*.

A hypermodule consists of a set of hyperslices and their integration relationships. Integration relationships stem from the composition rules of Subject Oriented Programming (SOP) [HO93]. In Hyper/J, a compositor tool can be used to associate integration relationships, therefore one can think of a hypermodule as a combination of hyperslices. Hypermodules can be used to encapsulate many types of software artefacts, components or whole systems. Each hyperslice is self-contained but with integration rules the system can work together in a co-ordinated fashion. The following code is of a hypermodule; a hypermodule contains a set of hyper slices and an integration relationship among the hyper slices.

```
hypermodule SEE_With_Display_And_Check
hyperslices: Feature.Kernel, Feature.Display
             Feature.Check
relationships:    mergeByName
```

In this hypermodule the relationship is “mergeByName” this means that units in the different concerns are related if they have the same name. The are other types of relationships within Hyper/J for example: merge (merge relationship between specified set of units), nomerge (opposite to merge, causes unit which match not to merge) and override (Indicates that one unit overrides another unit).

Implementation

Hyper/J is a tool that realises the hyperspaces approach from Java ^{TM}. The units that are supported are: packages, interfaces, classes and members. The concern matrix can work with these units and there is the ability to make hyperslices from these units. Integration of the hyperslices is possible to form hypermodules that can be generated into Java class files.

Relevance to AspectC#

Hyper/J uses a different approach but there are still lessons to be learned from it, a feature of Hyper/J that is of particular interest is that Hyper/J offers modularisation of crosscutting concerns without explicit language extensions. By making concerns declaritively complete they become self contained. Hyper/J does not make any language extensions.

2.5 .NET AOP Approaches

In this section we cover the principal .NET approaches to AOP. Although there exists many Java base AOP tools, the development of .NET tools is still only in nascent stages. We will cover two of the foremost approaches CLAW[Lam] and AOP# [AOP]. Other approaches include Aspect.NET [Mica] and IL based approaches like “Aspect-Oriented Infrastructure for a Typed, Stack-based, Intermediate Assembly Language” [DEC].

2.5.1 Cross Language Aspect Weaving (CLAW)

CLAW has many similarities to AspectJ as covered in section 2.4.1. CLAW borrows many features such as advice (before, after) and contains a similar Join Point model. But they dramatically differ in the process of aspect weaving. CLAW contains an Execution Engine (EE) that weaves base and aspect code together at runtime; the aspect to base code mapping is achieved through *weaving instructions* that are generated by a weave compiler. All weaving occurs at runtime and it is a future goal that the CLAW will be able to emit *'aspect assemblies'*. Any CLR compatible compiler can provide aspect assemblies, which in turn provide the aspect code to the Execution Engine. Aspects can be weaved with target methods that are not written in the same language thereby performing true cross language aspect weaving.

Architecture

The CLAW architecture consists of:

- A set of base assemblies.
- A set of aspect assemblies.
- *Weave definition files* that contains the mapping of aspect code to base code.

The base and aspect code can be written in any normal .NET language; it is then compiled using a compliant compiler into an assembly. The mapping from aspect to core methods is through two weave definition files: *weave.xml* and *weavemanifest.xml*. The weave.xml definition file contains a set of weaver definitions; each definition describes how a join point is mapped to an aspect. The weavemainfest.xml file contains the aspect or base assemblies. The weave compiler compiles the weave definition files into a weave instruction file that is used by CLAW to perform weaving at runtime. The following code segment shows the two weave definition files.

```
<weave>
  <aspect pointcut="App.Main"
```

```
          adviceType="before"
          advice="MyAspect.Before" />
</weave>
weave.xml

<manifest>
  <aspects>
    <assembly path="c:\temp\HelloWorld\MyAspect.dll"/>
  </aspects>
  <targets>
    <assembly path="c:\temp\HelloWorld\App.exe"/>
  </targets>
</manifest>
weavemanifest.xml
```

CLAW operates by weaving aspect code prior to JIT compilation of the target method. CLAW reads the IL of the target method and constructs an abstract syntax tree (AST). It then weaves in aspect IL and writes the modified IL into memory.

The following is a simple aspect using CLAW. It prints out “Hello from C# Aspect” before the target method is executed, the target method is specified in the weave.xml file in this case the method is MyAspect.Before. Join Points are specified in the weave.xml file.

```
public class MyAspect
{
  private static MyAspect this_;

  static MyAspect() { this_ = new MyAspect(); }

  public static MyAspect AspectOf() { return this_; }

  public void Before( CodeXP.CallContext c )
  {
    Console.WriteLine( "Hello from C# Aspect" );
  }
}
```

Summary

CLAW has not been released yet, but promises to be an extremely useful tool from the start. A current limitation is the security issue. CLAW allows developers to dynamically alter the behaviour of code within the CLR. Until a security policy can be created, this is a major limitation. Also CLAW cannot weave unmanaged and managed code, that is, code that runs outside the CLR. Future work includes support for intra-method join points, XPath syntax for join point selection and support for link-time weaving.

Relevance to AspectC#

One of CLAW's features is the ability to perform cross language weaving, VB.NET code can be weaved with C# code and vice vearsa. It was interesting finding out how CLAW performs this weaving, CLAW targets IL while AspectC# targets source code therefore it is not possible to gain an advantage here but the XML mapping technique is similar to the one employed by AspectC#.

2.5.2 AOP#

AOP# has been developed by Schupany, Schwanninger and Wuchner from the Siemens Corporation. It is named AOP# in accordance with the language C#. The requirements of AOP# are:

1. No language extensions; the user should not have to learn any new constructs and a standard compiler is used.
2. Complete separation of base and aspect code.
3. Easy mechanism to join base and aspect code; something along the lines of the J2EE EJB container model.
4. The ability to switch on and off aspects. This concept is called *"aspectual polymorphism"*.

A feature of AOP# is that any language supported by the .NET framework may be used as either base or aspect code.

Architecture

AOP# architecture consists of three parts:

1. *Application assemblies* - holds the application or business code called '*core code*'.
2. *Aspect assemblies* - holds the aspect code.
3. Coordinating component called the *XML-Connector*. The AOP environment use the connector to associate aspects with the core code.

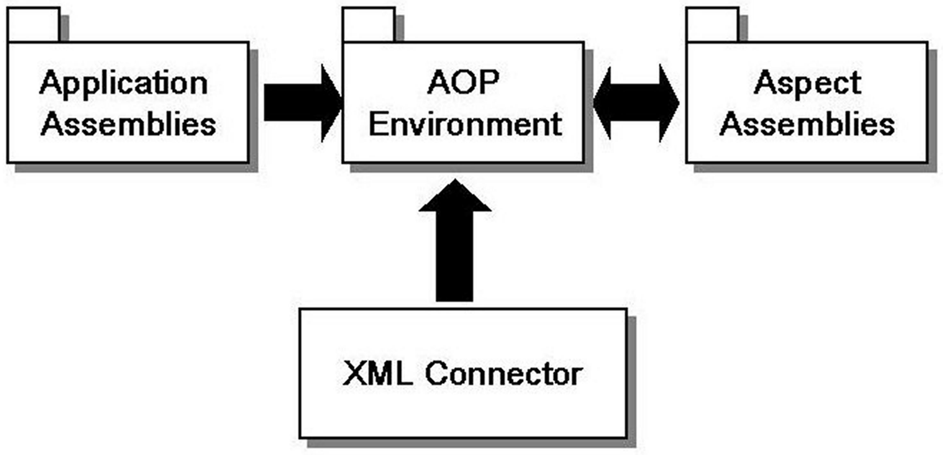

Figure 2.8: Architecture AOP#.

Base and aspect classes are compiled into MSIL and then composed into assemblies, from there the connector advises the AOPEnvironment which core classes should have aspects. The AOPEnvironment intercepts at runtime the core code and inserts the aspect code. This interception is achieved using the Profiling and Metadata API of the .NET framework [Micd]. One restriction with this approach is that only managed code may be used and the Profiling API cannot handle unmanaged code. An example of unmanaged code is memory allocation (pointers) in C++ as this cannot be mapped to MSIL.

Aspects in AOP# are ordinary .NET classes that are declared as abstract and inherit from a base class called `Aspect`. Aspects are organised into assemblies and there are no language extensions. The implementation of an aspect consists of:

- The (abstract) declaration of an interface expected from any core class the aspect should later be weaved into.
- The implementation of the aspects method, this corresponds to the "around" advice in AspectJ.

The expected interface of the aspect consists of abstract methods that can be divided into two categories marked by attributes *"isRequired"* and *"IsExtended"*.

- IsRequired methods aim to interact with 'aspectised' core classes. It is similar to the declarative completeness of Hyper/J.
- IsExtended methods aim to extend core class methods. They require a concrete implementation of the form Extend_*, where * is the method name in the core class.

The following is a simple example of an aspect in AOP#. It represents a simplistic bounds checker that works on graphical elements of a core system.

```
public abstract class BoundsChecker : Aspect
{
   private const int MAX_X = 100;
   [IsRequired]
   public abstract int GetX();
   [IsRequired]
   public abstract void SetX();
   [IsExtended]
```

```
      public abstract int ShiftX(int shiftFor);

      //Implementation of aspect
      public void Extends_ShiftX(int shiftFor)
      {
         //before
         bool isOutOfBounds ;
         if(GetX() + shiftFor > Max_X)
         {
            SetX(Max_X - shiftFor);
         }
         //call to core method
         ShiftX(xValue);
         //after
         ..
      }
   }
```

The XML-connector describes how core classes and aspect methods should be mapped together.

Aspectual Polymorphism

Aspectual Polymorphism relates to the ability of turning on and off aspects at run-time. During the course of the application the aspect context (set of aspects being aspectized) can change. This is a ne feature and not seen in any other AOP tool.

Summary

A first prototype of AOP# is due for release, but because all concepts are taken from existing successful approaches the designer expect it to be useful from the beginning. AOP# and CLAW are quite similar in their approach to weaving aspects and working on assemblies, a key difference is the aspectual polymorphism were aspects can be turned on and off.

Relevance to AspectC#

Both AOP# and CLAW are very similar in their approach to AOP, but what separates them is the Aspectual Polymorphism in AOP#. This is interesting to AspectC#

and it would be interesting to see if this feature can be added to AspectC#.

2.5.3 Other .NET approaches

As stated .NET tools are still in early stages of development but two other promising approaches are Aspect.NET[Mica] and MSIL-based AOP tool by Dechow [DEC]. Aspect.NET aims to research AOP tools that target both source code and MSIL. It aims to create tools that are both static and dynamic. They also intend to investigate language extension to the C# language (similar to AspectJ extensions to Java). Dechow's tool aims at weaving on MSIL and can be describe as *"source to-source preprocessor for the Common Intermediate Language (CIL)"*[DEC].

2.6 Summary

In this chapter we have covered a state of the art review of the .NET platform and the main Java and .NET approaches to AOP. But there exist many other approaches to AOP including supported systems like AspectR[ASPb], AspectS[Hir], DemeterJ[Lie] and research prototypes like AspectC[Coa] and AspectC++[ASPa].

Chapter 3

Requirements and Design

3.1 Introduction

In this chapter, firstly we look at the requirements for AspectC#. We then discuss the design of the system with regard to other approaches mentioned in Chapter 2.

3.2 Requirements

One of the objectives of the project is to develop a tool ‘AspectC#’ that enables AOP constructs within the C# language. The system must be developed within a 12-week time frame and a small example must also be developed to show a working prototype.

The major use of the system will be to apply AOP constructs within the C# language. Developers should be allowed to use a rich set of instructions that will aid in modularisation of crosscutting concerns. AspectC# can be applied to a new project in development or to existing applications by refactoring the design of the application.

3.2.1 Functional Requirements

AspectC# must allow developers to use AOP mechanisms within C#. It must allow developers to specify aspect and base classes, and to allow the merging of these classes and create ‘aspect assemblies’, which are normal assemblies and can run under the CLR.

1. AspectC# must allow the developer to modularise crosscutting concerns within the C# language.

2. The tool should not make any language extensions to the C# specification [Micb]. The reasoning behind this requirement is that the developer does not have to learn new language constructs making it easier to learn AOP techniques. Also the fact that there are no language extensions means that a standard C# compiler can be used. This reduces the development time of the tool and allows the use of GUI IDEs such as Visual Studio .NET (VS.NET) as a development environment.

3. Although we see the prototype as a proof of concept; a requirement is that the system should be extensible and may, in future versions of the system, be able to support multiple languages rather than just C#. The semantics of AspectC# should also be extensible allowing for new constructs if necessary.

4. The system must not permanently modify the base code. AspectC# must take base source classes as inputs and perform weaving inside the system, leaving the base classes unchanged. The reason behind this is that with AspectC# it should be possible to compile and run base classes on their own, and when using AspectC# the base code developer can be oblivious to the aspect class and concentrate on the functionality of the base concern. A temporary directory of intermediate weaved files should be produced to enable debugging purposes.

5. AspectC# must be easy to use and have a small learning curve. It should be based on the ‘state of the art’ of AOP tools and technologies to try and make the tool effective from the beginning.

6. The evaluation of the project is an extremely important requirement. Firstly the resulting weaved code must be compared with an object oriented version of the example system; and the system must be compared with other AOP tools on: How the system specifies aspects, what composition mechanisms the system provides, implementation mechanisms of the system, decoupling issues, software process and usability must be evaluated.

3.2.2 Non-Functional Requirements

The non-functional requirements are that the tool should be developed using Microsoft Visual Studio .NET (Release Version) and should be able to run on the Microsoft .NET Framework ^{TM}.

Requirements Summary

In summary the requirements of the system are:

- Enable the modularisation of crosscutting concerns within C#.
- No language extensions to C#.
- System should be extensible for future development.
- Complete separation of base and aspect code.
- Ease of use.
- Thorough evaluation.

3.3 Design

This section attempts to describe the design of the system. Firstly we begin by comparing and contrasting the state of the art approaches in AOP tools both in Java and .NET. Finally we describe in detail the AOP Engine, which satisfies requirement no.1 (which is made up of modules): Parser, AST Representation, which satisfies requirement no.2 and no.3, Aspect Mapping, which satisfies requirement no.4, Weaver satisfies requirement no.4 and Compiler.

3.3.1 Different Approaches

There exist many approaches to AOP, the principle ones have been discussed in section 2.4. Figure 3.1 highlights where each currently available tool applies its composition mechanism. We evaluated each approach based on the language that it used, the architecture of the system and how the system was implemented.

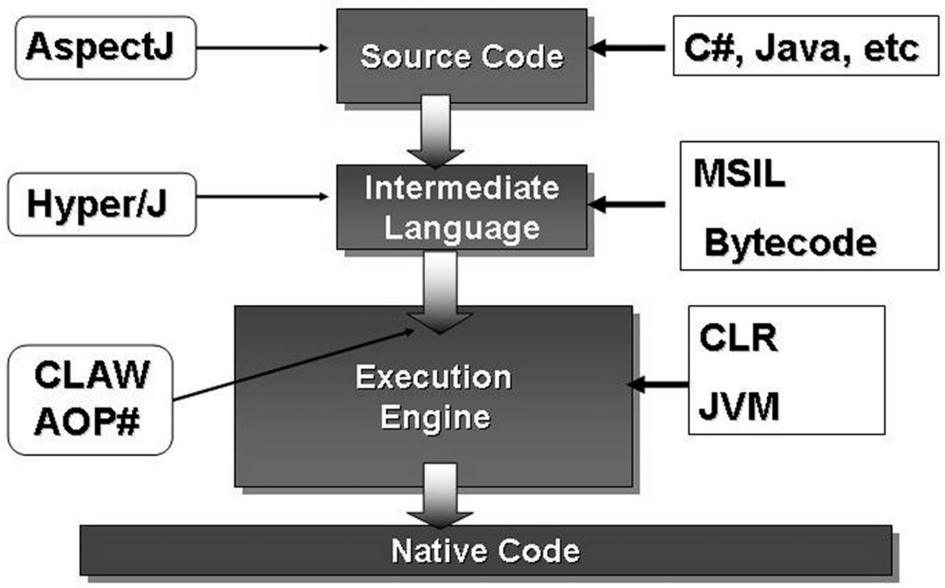

Figure 3.1: Different AOP tool approaches.

AspectJ [Tea] covered in section 2.4.1 works with source code and weaves aspects at compile time; it has a lot of specific constructs and is probably the most used tool in the field. Hyper/J [HYP] covered in section 2.4.2 stems from subject oriented programming and does not contain the idea of aspect and base but rather a concern and concern interaction. Hyper/J works on the idea of declarative completeness were everything a concern uses is declared in the class. Hyper/J weaves concerns at a byte code level.

CLAW [Lam] covered in section 2.5.1 has similar syntax to AspectJ and the central idea of an aspect. CLAW weaves base and aspect code at a MSIL level and then calls the JIT compiler. CLAW uses a XML mapping of aspects to base code. AOP# [AOP] covered in section 2.5.2 again works on a MSIL level and uses an XML mapping of aspect to base code.Table 3.1 summaries each approach.

3.3.2 Our Approach

The approach that we decided to take was to use similar semantics and architectures of existing tools and use the features from these other tools for our approach. By using similar methods of existing successful tools we can infer that hopefully our tool will be useful from the beginning. But it must be stated that this tool is a proof of concept and a goal of further research is to make it the leading AOP tool for the .NET framework.

In our approach it was decided to use similar semantics to AspectJ [Tea]. That is, we make a clear separation of what is base (often describe as ‘business’ or ‘core’ code) and what is aspect code. The Join Point model is a sub set of the AspectJ Join Point model. An aspect in AspectC# is the same as in AspectJ but it must be declaratively complete like concerns in Hyper/J. We use an XML base mapping of

Topic	AspectJ	Hyper/J	CLAW	AOP#
Language(s) Used	Works with the Java Language.	Works with the Java Language.	Works with any .NET compatible language.	Works with any .NET compatible language.
Expressive Power	AspectJ contains constructs for declaring aspects and pointcuts.	Hyper/J contains constructs for identification and encapsulation of concerns.	CLAW uses aspects and a xml mapping.	AOP# uses aspects and a xml mapping.
Composition mechanisms	AspectJ has explicit language extensions. Aspects are Explicit applied to classes.	Hyper/J has no language extensions. Concerns composed using classes.	CLAW has no language extensions. Aspects are applied to classes.	AOP# has no language extensions. Aspects are applied to classes.
Implementation mechanisms	AspectJ is applied to source code. Composition is static at compile time.	Hyper/J is applied to byte code. Composition is static at compile time	CLAW is applied to MSIL within the CLR. Composition is dynamic at run time.	AOP# is applied to MSIL. Composition is dynamic at run time.
Ease of Use	AspectJ is extremely user friendly with IDE developer plug-ins also available.	Hyper/J is more a complex approach but does have GUI support.	Currently no prototype available.	Currently no prototype available.
Aspectual Polymorphism	No support	No support	No support	Support for Aspectual Polymorphism

Table 3.1: Different AOP tool approaches

aspects to core classes.

The Language

AspectC# has a similar semantics to AspectJ, it supports two types of crosscutting implementation:

- **Static**: Enables the developer to add variables and methods to existing types
- **Dynamic**: Enables the developer to define additional implementation to run at well defined points in the program.

Aspect Mapping (Join Point Model)

AspectC# allows for advice to be associated with base methods before a method execution and after a method execution. This is a sub set of the AspectJ Join Point Model. As we did not want any language extension we use an XML mapping of aspect advice to base code, this is instead of the Pointcut designators in AspectJ.

Aspect

An aspect is a modular unit of crosscutting concerns and it contains advice. Advice has the same meaning as in AspectJ, but as there are no language extensions to C# so the advice must be proper C# code. Aspects are declaratively complete therefore any variable or method a piece of advice uses must be declared within the advice or as an abstract method or variable. There is also the reflective access of a join point available to the developer. The reflective access allows the developer to get information about the current join point and to access data about the current join point which may be useful in the development of crosscutting concerns. In summary, the semantics of AspectC# are:

- An *Aspect* is a modular unit of crosscutting concerns, it contains advice that can be applied to base code. Aspects must be declaratively complete.

- *Advice* can be of type *before, after* or *around*, as covered in section 2.4.1. It is a method like mechanism used to insert code at certain well defined pointed in the program execution.

- The Join Point model is a subset of the AspectJ Join Point model. A file called the *Aspect Deployment Descriptor* is used to hold these mappings.

- The developer has reflective access to a join point, with use of a special variable called `ThisJoinPoin`.

In Table 3.1 we summarised the different approaches of the most popular available tools at present; Table 3.2 summarises our approach.

Issue	AspectC#
Language(s) Used	AspectC# works with C# and hopefully in future versions it will extend to all .NET languages.
Composition Mechanisms	AspectC# has a similar semantics to AspectJ, it contains the constructs for declaring aspects, implementing advice and associating aspects with base classes in the form of the Aspect Deployment Descriptor.
Expressive Power	AspectC# contains support for constructing and declaring aspects. It uses a subset of the AspectJ Join Point model and an XML mapping.
Implementation Mechanisms	AspectC# has no explicit language extensions. Aspects are applied to classes, with the aid of the Aspect Deployment Descriptor.
Implementation Mechanisms	AspectC# is applied to source code. Composition is static at compile time.
Ease of Use	It is a goal of AspectC# that it will be easy to use. It is based on current technology and hopefully will have a low learning time.
Aspectual Polymorphism	No support.

Table 3.2: Aspect C# approach

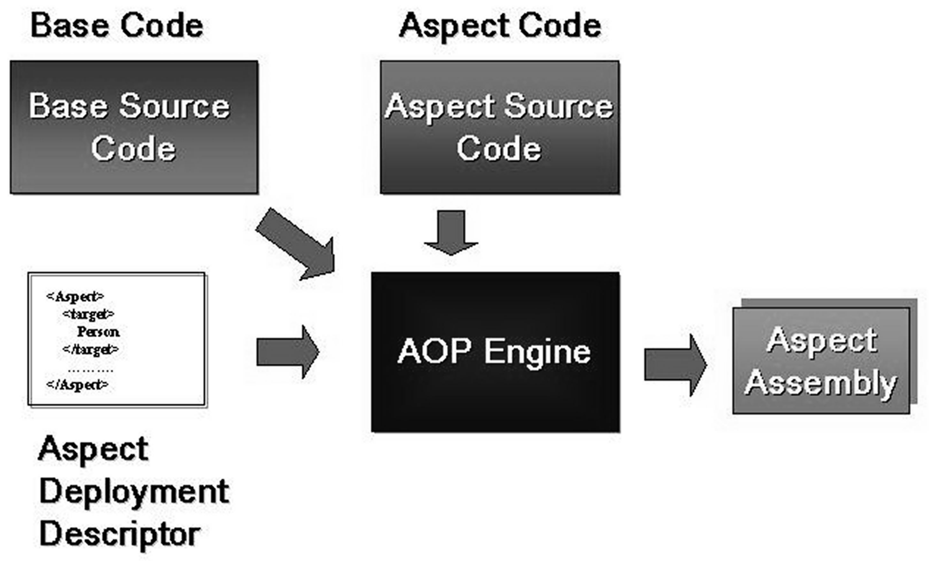

Figure 3.2: Project Architecture.

3.3.3 Project Architecture

The project architecture, as shown in figure 3.2 works with base and aspect source code. Therefore it needs a parser to analyse the source code. Next it must model this code in the form of an Abstract Syntax Tree (AST), by using the CodeDOM API [Micc]. The weaver merges base and aspect graphs\ trees. Finally the compiler compiles the tree using a standard C# compiler. The AOP engine consists of five main areas, as shown in figure 3.3. The 5 components are described in the subsequent sections:

- Parser: Parsing of both base and aspect code.

  Satisfies requirement no.1

- AST: Representation of source code files as an AST.

  Satisfies requirement no.1, no.2, no.3

- Aspect Mapping (Join point): The Join point model.

  Satisfies requirement no.1, no.4

- Weaver: Weaving aspect and base graphs together.

  Satisfies requirement no.1, no.4

- Compiler: Compilation of weaved aspect and base graph.

  Satisfies requirement no.1

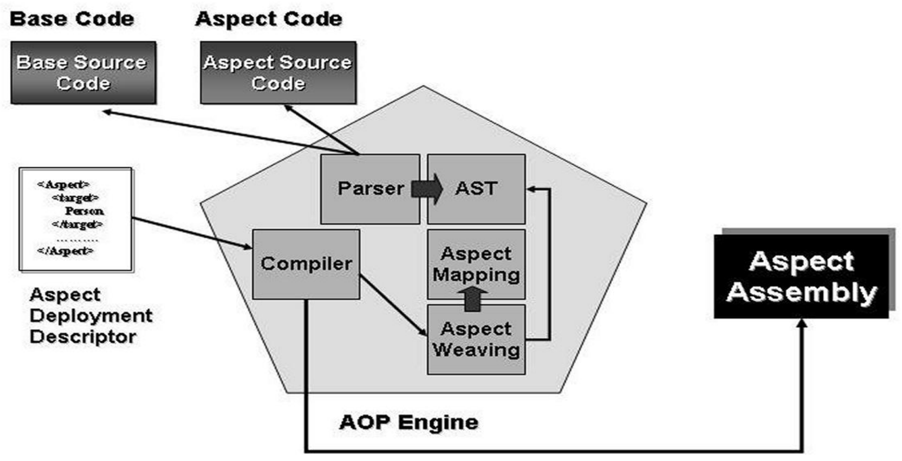

Figure 3.3: AOP Engine Architecture.

Parser

The purpose of a parser is to analyse the phrase structure of a program. As we decided that the weaving would only occur at well defined join points (i.e. within class and member functions) this effected the requirements of our parser. We only needed a subset of the C# grammar for the prototype. As shown in figure 3.4 our parser can model namespace, class, function and variable items. The classes are discussed in detail in section 4.2.

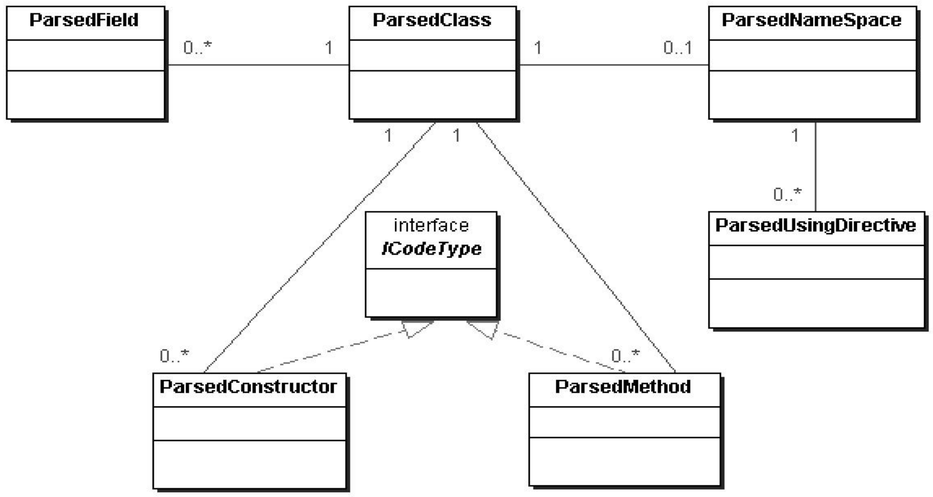

Figure 3.4: Parser Class Diagram.

AST Representation

Once the source file has been parsed the next step is to create an AST representation of the code. By using the CodeDOM API as covered in section 2.2.3, we have a language neutral AST. We believe that by using the AST in future versions of our system, it will be possible for cross language aspect weaving because other languages can be modelled as a CodeDOM object graph and weaving can occur at the object graph level.

Aspect Mapping

The Join Point model is a significant part of any AOP tool; we decided to use a sub set of the AspectJ Join Point model. An XML based approach for mappings was used, where aspects can be mapped to base methods. The following code segment is the Document Type Definition (DTD) for the Aspect Deployment Descriptor.

```
1.  <!ELEMENT Aspect(TargetBase, AspectBase, AspectMethod+,
                  Target+)>
2.  <!ELEMENT TargetBase (#PCDATA)>
3.  <!ELEMENT AspectBase (#PCDATA)>
```

```
4.  <!ELEMENT AspectMethod(Name, Namespace, Class, AMethod)>
5.  <!ELEMENT Target (Namespace, Class, Method+)>
6.  <!ELEMENT Name (#PCDATA)>
7.  <!ELEMENT Namespace (#PCDATA)>
8.  <!ELEMENT Class (#PCDATA)>
9.  <!ELEMENT AMethod (#PCDATA)>
10. <!ELEMENT Method (Name, Type, Aspect-Name)>
11. <!ELEMENT Type (#PCDATA)>
12. <!ELEMENT Aspect-Name (#PCDATA)>
```

The structure of the Aspect Deployment Descriptor is quite simple. An `Aspect` element consists of four main elements: `TargetBase`, `AspectBase`, `AspectMethod` and `TargetMethod`.The `TargetBase` and `AspectBase` elements represent the configuration information, where the base and aspects classes are found, as shown on lines 2 and 3. An `AspectMethod` represents an aspect method it consists of a name, namespace of the class, class name and the method name, as shown on line 4. A `TargetMethod` represents the target or base class it consists of namespace, class name and method name, as shown on line 5. A `Method` element represents the target method it consists of name, type of advice to apply (e.g. before, after or around), and the aspect method name that contains the actual advice, as shown on line 10.

Aspect Weaver

The weaving involves making sure that the advice specified by the XML Aspect Deployment Descriptor is inserted into the target methods. The weaving covers both static crosscutting called '*introduction*' and dynamic crosscutting.

Variables and methods can be introduced by using an attribute `[introduction(<string>)]`, where `<string>` is a list of classes that the variable or method should be introduced too.

With dynamic crosscutting advice is specified by the XML Aspect Descriptor.

Target methods are identified within the XML file.

As shown in figure 3.5, the AOP engine has explicit access to member functions and variables through the CodeIdentifier class.

Within the weaving module aspect conflicts are resolved, if more than one aspect is applied to a base method then there needs to be an ordering of aspects. Aspects are applied to base methods depending on the type of advice:

1. Any `around` advice is firstly applied.

2. `Before` advice is applied, if more than one piece of before advice is applied to a base method then the order in which it appears in the Aspect Deployment Descriptor is the order in which it is applied.

3. `After` advice is then applied, again if more than one piece of after advice is applied to a base method then the order in which it appears in the Aspect Deployment Descriptor is the order in which it is applied.

With AspectC# aspects can be tailored to the developers needs. An aspect can be general and affect all classes in an application or they can be specific and only affect a small number of classes. With runtime based approaches to AOP many tools have problems with the security issue and injecting code before JIT compilation, as aspect weaving is performed at compile time there are no security issues with AspectC#.

Compiler

The final job of the AOP engine is the compilation of weaved source code into an assembly; we call the compiled assembly an ‘aspect assembly’. Here a standard C# compiler can be used because there is no language extension.

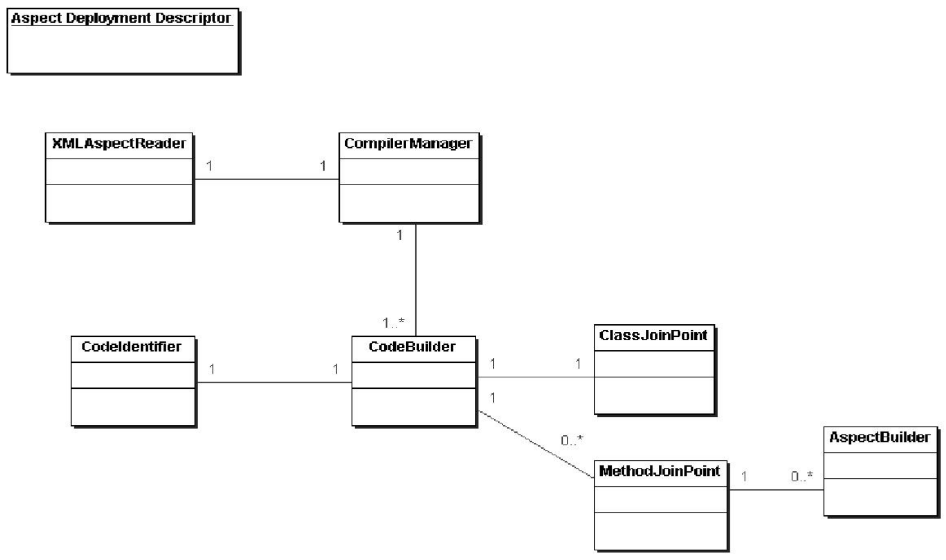

Figure 3.5: Compiler Class Diagram.

3.4 Summary

In this chapter we have covered the main functional and non-functional requirements of the system. The usage of the tool has been described and finally the design of the tool has been discussed. AspectC# design allows developers to capture crosscutting concerns in a modular way. There are no language extensions necessary and aspect and base code is kept separate from each other. We believe our design is modular and extensible so that in future versions of the tool it may be possible for cross language aspect weaving.

Chapter 4

Implementation

4.1 Introduction

This chapter describes the implementation of the prototype system. We describe the AOP Engine and its parts: Parser, AST Representation, Weaver, Compiler and Aspect Deployment. Finally there is a summary of the implementation. The system was developed using the Microsoft Visual Studio .NET release version. The database used for the example project was the Microsoft SQL server version 8.0.

4.2 AOP Engine

The AOP Engine forms the heart of AspectC#, as shown in figure 3.3 it consists of five modules. Each module is responsible for a certain area of the overall requirement of allowing the modularisation of crosscutting concerns.

4.3 Parser

This section describes the implementation of the parser of the system. We could have used a parsing tool like YACC [YAC], LEX [LEX] or Visual Parse [SAN]; but it was decided that our system did not warrant the additional complexity and dependencies on 3rd party libraries or software so a simple parser was developed that parsed the necessary constructs namely, class, variable and member level.

As discussed in section 3.3.3, the main functionality of the parser is to lexically analyse the source file whether it is a base or aspect source file. The parsing involved using regular expressions classes from the namespace `System.Text.RegularExpressions`. Regular expressions provide a mechanism for capturing and matching large amounts of text. The `RegularExpressions` namespace is based upon the PERL5 [Per] regular expression syntax.

The classes involved in the parsing are, see figure 3.4:

- `CodeIdentifier`: represents a parsed source field with references to class, field and member information.
- `ParsedNameSpace`: represents a parsed `namespace` directive.
- `ParsedUsing`: represents a parsed `using` directive.
- `ParsedClass`: represents a parsed `class` directive.
- `ParsedField`: represents multiple parsed `variable` directive.
- `ParsedConstructor`: represents a parsed `constrcutor` directive.
- `ParsedMethod`: represents a parsed `method` directive.
- `ParsedParameter`: represents a parsed `parameter` directive.
- `ClassRegex`: contains the regular expressions.

As an example of parsing the following code segment represents the process involved in parsing a `namespace` directive. Firstly the C# grammar for a namespace [Micb] is inspected to find the possible combinations. Then the regular expression is generated and finally the code may be using within the `CodeIdentifier` class. We begin by looking at the namespace grammar as defined in the C# specification.

```
compilation-unit:
using-directivesopt  global-attributesopt   namespace-
member-declarationsopt

namespace-declaration:
namespace   qualified-identifier   namespace-body   ;opt

qualified-identifier:
identifier
qualified-identifier   .   identifier

namespace-body:
{ using-directivesopt   namespace-member-declarationsopt }

using-directives:
using-directive
using-directives   using-directive

using-directive:
using-alias-directive
using-namespace-directive

using-alias-directive:
using   identifier   =   namespace-or-type-name   ;

using-namespace-directive:
using   namespace-name   ;

namespace-member-declarations:
namespace-member-declaration
namespace-member-declarations   namespace-member-declaration

namespace-member-declaration:
namespace-declaration
type-declaration

type-declaration:
class-declaration
struct-declaration
interface-declaration
enum-declaration
delegate-declaration
```

This grammar segment is the full grammar for the `namespace` declaration in the C# specification. The next code segment represents the regular expression for the namespace grammar.

```
\s?(namespace)(\s\w+)(\u002E\w+)*

where,
\s?           = zero or more whitespace characters.
(namespace)   = the text namespace (all lower case).
(\s\w+)       = any whitespace character followed by
                one or more letters.
(\u002E\w+)* = a fullstop followed by one or more letters,
                repeated zero or more times.

E.g. namespace Ie.Tcd.AspectCSharp;
```

4.4 Abstract Syntax Tree

The AST representation of the source code is vital to the project, a requirement covered in section 3.2 is that the system should be extensible. By modelling the source code as a `CodeDOM` tree we keep it independent from a specific language; this will leave the possibility of cross language weaving. The AST representation relies heavily on the CodeDOM API as discussed in section 2.2.3. The CodeDOM API provides interfaces, classes and architecture to represent the structure of a source code document independent of a programming language. The classes involved in the representation of the AST of a source code file are:

- `CodeBuilder`: The CodeBuilder class acts a the central location for the creation of the AST. It uses the `ParsedNameSpace`, `ParsedUsing`, `ParsedClass`, `ParsedMethod`, `ParsedField` classes from the parsing module to create the tree.

- `ParsedNameSpace`, `ParsedUsing`, `ParsedClass`, `ParsedMethod`, `ParsedField` classes: These classes are used to generate the information about the source file.

The process of representation of source code as an AST is best shown in the form of a sequence diagram. Figure 4.1 shows the process of source code to AST representation.

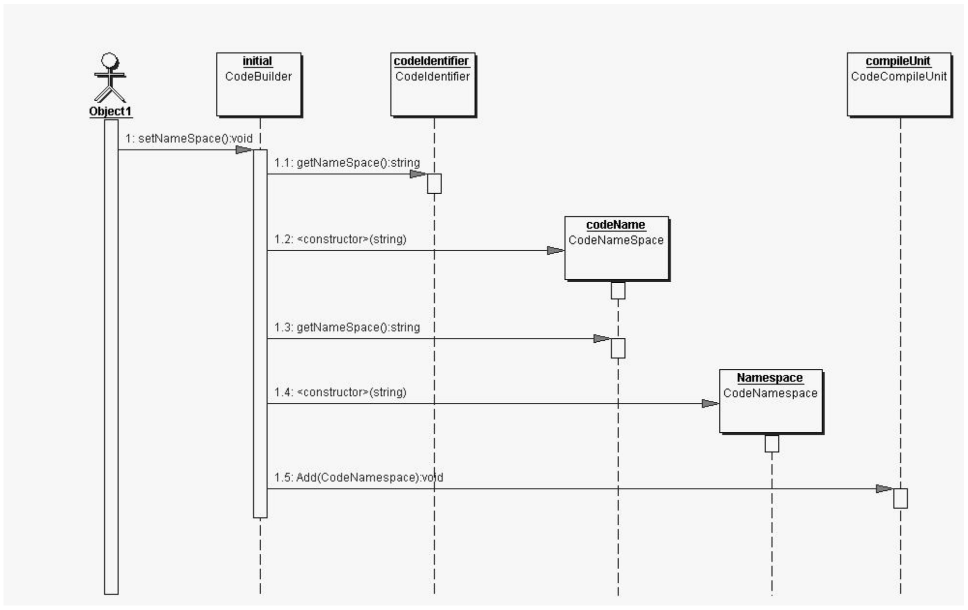

Figure 4.1: Sequence Diagram of AST representation

4.5 Aspect weaver

The merging of aspect and non-aspect code is a critical element in the design of any aspect-oriented language mechanism. The aspect weaving is performed on two types of crosscutting implementations as stated: dynamic, **advice** and static, **introduction**. In section 3.3.3 we discussed the design our aspect weaving mechanism, figure 3.5 illustrates the structure. The classes involved in weaving are:

- `CodeBuilder`: Models the source code file as an AST, performs weaving at methods joinpoints.
- `ClassJoinPoint`: Each source file\ class contains one ClassJoinPoint, which contains target class information and aspect information.

- `MethodJoinPoint`: Each method is checked to see if it has associated advice, if this is true then a MethodJoinPoint object is created that contains advice for the method.

- `Advice`: Conatins the type of advice it is (i.e. before, after, around) and a reference to an spectBuilder object that represents the aspect code to be woven at the join point.

- `AspectBuilder`: An AspectBuilder object contains explicit access to the fields and methods of an aspect class. An Advice object queries the Aspect-Builder to obtain the advice code.

4.5.1 Dynamic Crosscutting Implementation

Advice from aspect classes is woven with base classes by firstly identifying the target methods (with Aspect Deployment Descriptor) and then inserting the advice into the AST object graph. Figure 4.2 shows the sequence diagram of weaving aspect advice into target methods.

The following code snippet shows the weaving of aspect and base code for before and after advice. Each method is firstly checked to see if has advice using the `hasAdvice` method, this method contains access to the Aspect Deployment Descriptor information and this in turn can be used to identify methods that have advice. If the method has advice then a `MethodJoinpoint` object is created that holds the specific advice for the given method. Finally the weaving is performed at the object graph level.

```
MethodJoinPoint methodJP = hasAdvice(c.getNameAndParameters());
if(methodJP.hasAdvice())
{
  if(before)
  {
    foreach(Advice beforeAdvice in beforeList)
    {
      AspectBuilder aspectAdvice =
```

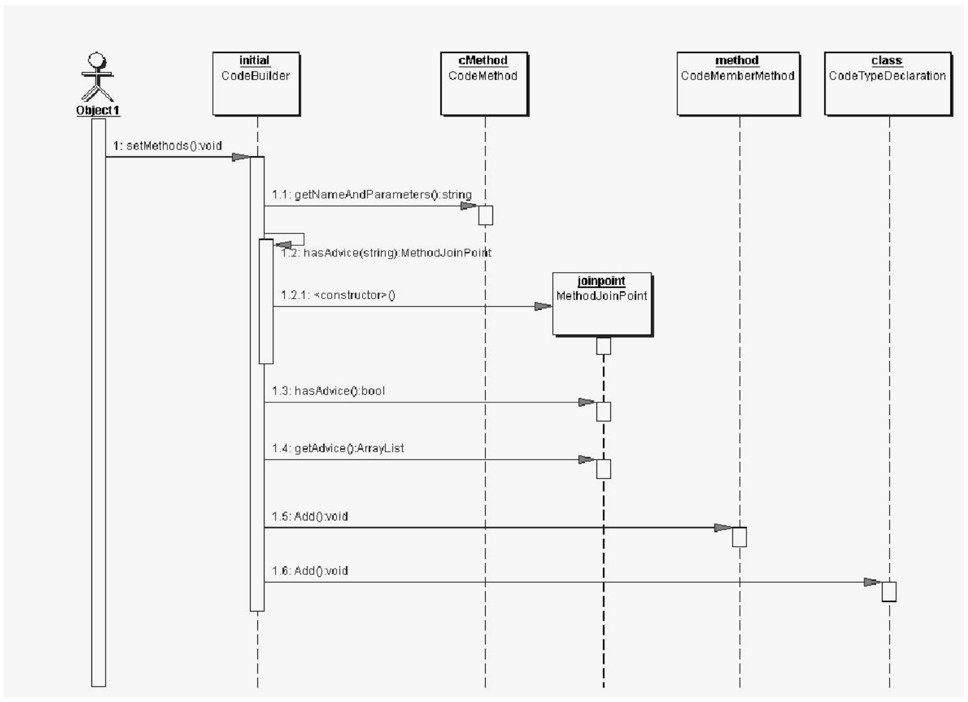

Figure 4.2: Sequence Diagram of Aspect Weaver

```
    beforeAdvice.GS_Aspect;
    string codeBefore = aspectAdvice.
                  getAspectCode(beforeAdvice.GS_methodName);

    CodeExpressionStatement expressionBefore = new
        CodeExpressionStatement(
           new CodeSnippetExpression(codeBefore));

    method.Statements.Add(expressionBefore);
  }
}

string code = c.getFinalText();
CodeExpressionStatement expression = new
        CodeExpressionStatement(
           new CodeSnippetExpression(code));

method.Statements.Add(expression);

if(after)
```

```
  {
    foreach(Advice afterAdvice in afterList)
    {
      AspectBuilder aspectAdvice = afterAdvice.GS_Aspect;
      string codeAfter= aspectAdvice.getAspectCode(
                         afterAdvice.GS_methodName);
      CodeExpressionStatement expressionAfter = new
          CodeExpressionStatement(
              new CodeSnippetExpression(codeAfter));
      method.Statements.Add(expressionAfter);
    }
  }

  Class.Members.Add(method);
}
```

4.5.2 Static Crosscutting Implementation

In order for introduction to be enabled within AspectC# it was necessary to use the attribute mechanism of the C# language. An attribute is an object that represents data that you would like to associate with an element within your program [Lib01]. We created a custom attribute called `introduction`. The following code segment shows the code of the creation of the attribute.

```
[AttributeUsage( AttributeTargets.Field |
                 AttributeTargets.Method)]
public class IntroductionAttribute : System.Attribute
{
   private string Name;

   public IntroductionAttribute(string Name)
   {
      this.Name = Name;
   }

   public string getName()
   {
      return this.Name;
   }
}
```

Once the `introduction` attribute was created we could associate the attribute with elements within the program. The next code segment demonstrates the usage of Introduction by adding the method `sayHello` into the class HelloWorld. On line 1 we can see by using the [Introduction] attribute we can specify we want the following method to be added to the target class.

```
1.  [Introduction("HelloWorld")]
2.  public void sayHello()
3.  {
4.    Console.WriteLine("Hello Howard!");
5.  }
```

Within the AOP Engine the CodeBuilder class analyses the aspect class for any instances of introduction. If any are found then the variable or method is added into the CodeDOM object graph. The following code shows the procedure involved for introduction.

```
foreach(AspectBuilder aspect in getAspects())
{
   foreach(CodeMethod c in
           aspect.getIntroductionMethods(Class.Name))
   {
       introductionMethods.Add(c);
   }

   foreach(CodeField cf in
           aspect.getIntroductionFields(Class.Name))
   {
       introductionFields.Add(cf);
   }
}

//Add in CodeMethod and CodeField into CodeDOM object
//graph
```

4.6 Compiler

The `System.CodeDom.Compiler` namespace contains classes that can be used to manage compilation of source code from CodeDOM object graphs. The `ICodeCompiler` can be used by developers of generators and compilers. The `Microsoft.CSharp.CsharpCodeProvider` class provides an implementation of the C# compiler. The `System.CodeDom.Compiler.CompilerResult` class gives access to any errors or warnings generated during the compilation process. The follow code segment shows the internal compilation process within the AOPEngine.

```
Microsoft.CSharp.CSharpCodeProvider csharp = new
              Microsoft.CSharp.CSharpCodeProvider();
ICodeCompiler cscompiler = csharp.CreateCompiler();
CodeCompileUnit[] units = getCompileUnits(list);
CompilerResults compresult =
       cscompiler.CompileAssemblyFromDomBatch(compparams,
                                              units);

if ( compresult == null || compresult.Errors.Count > 0 )
{
  string errors = "";
  foreach (CompilerError err in compresult.Errors)
  {
    errors += err.ToString() + "\n";
  }

  Console.WriteLine("Errors: " + compresult.Errors.Count);
  Environment.Exit(1);
}
```

Currently the user only has access to a command line compiler tool. The user can specify a number of options:

- Output files.

  /out:<file>Output assembly name, must be fully qualified.

  /target:exe Build a console executable.

  /target:dll Build a library.

- Input files.

  /add:<file>Aspect Deployment Descriptor, XML aspect file.

  /main:<Main Class>Main class (Entry point of application) if exe.

With this compiler the developer can create either executables or libraries and specify where the Aspect Deployment Descriptor file is located.

4.7 Aspect Deployment Descriptor

The Aspect Deployment Descriptor is the name given to the aspect mapping mechanism. The `System.XML` namespace provides support for processing XML files. The `System.XML` namespace is part of the base class library. The Document Object Model (DOM) is what is used to model the XML files. A single XML file is read into AspectC#, which contains the mappings of aspects to targets and the base and aspect code directories.

The Aspect Deployment Descriptor acts as the main configuration file for the application. In the file the aspect and target directories are specified as well as the aspect to base code mapping. The aspect and target directories contain the actual C# files used in the merging of aspect and base code.The classes involved in the Aspect Deployment Descriptor are:

- `CompilerManager`: Main entry point of application, handles configuration of aspect and target files, as well as compilation.

- `XMLAspectReader`: Helper class for reading the aspect deployment descriptor.

The following is an example Aspect Deployment Descriptor:

```
<?xml version="1.0" encoding="utf-8" ?>
 <Aspect>
    <TargetBase>D:\Project\Test</TargetBase>
```

```
4.     <AspectBase>D:\Project\Test</AspectBase>
5.
6.      <Aspect-Method>
7.         <Name>AspectTest</Name>
8.          <Namespace>Test</Namespace>
9.          <Class>Test2</Class>
10.          <AMethod>before()</AMethod>
11.     </Aspect-Method>
12.
13.    <Aspect-Method>
14.          <Name>AspectTest2</Name>
15.          <Namespace>Test</Namespace>
16.          <Class>Test2</Class>
17.          <AMethod>after()</AMethod>
18.     </Aspect-Method>
19.
20.     <Target>
21.          <Namespace>Test</Namespace>
22.          <Class>HelloWorld</Class>
23.          <Method>
24.               <Name>SayHello()</Name>
25.               <Type>before</Type>
26.               <Aspect-Name>AspectTest</Aspect-Name>
27.          </Method>
28.          <Method>
29.               <Name>SayHello()</Name>
30.               <Type>after</Type>
31.               <Aspect-Name>AspectTest2</Aspect-Name>
32.          </Method>
33.    </Target>
34. </Aspect>
```

In this code listing lines 3-4 contain the configuration of the aspect and target directories. Lines 6-18 contain information about the aspect classes such as the name and actual C# class containing the aspect. Lines 20-33 contain information about the target classes and methods.

4.8 Summary

In this chapter we have looked at the implementation of AspectC#. We have examined the implementation of the five main areas of the AOP Engine: Parser, AST, Aspect Deployment Descriptor, Weaver and Compiler, collectively they enable the modularisation of crosscutting concerns. The design and implementation has been completed so that it is extensible and there are no language extension to C#.

Chapter 5

Evaluation

5.1 Introduction

In this chapter, we will evaluate both .NET platform as an AOSD platform and our tool AspectC#. Firstly we evaluate .NET as AOSD platform, next we describe our methods in the evaluation, and then we will evaluate it against the currently available AOP tools and methodologies. Finally there is an evaluation of a small example using AspectC#.

The main goal of AspectC# is the modularisation of crosscutting concerns. In the following sections we will evaluate AspectC# and other AOP tools based on how they specify aspects, the language the system used, how is it was implemented, the usability of the tool and the actual software process.

5.2 .NET as an AOSD platform

"A platform is any base of technologies on which other technologies or processes are built. In computers, a platform is an underlying computer system on which application programs can run." [is.]. AOSD is a new technology that enables the separation of concerns in software development. The question we would like to answer is: Does the .NET platform enable AOSD methods or techniques? We believe that the .NET platform does enable AOSD, we have built a tool AspectC#

that enables the modularisation of crosscutting concerns within C# without any language extensions. We believe the .NET platform can offer AOSD more than other previous platforms especially in the following areas: MetaData, Reflection, Code Generation and Cross Language Aspect Development.

5.2.1 Meta Data

Metadata is information about data, information about code, types, assemblies, etc. that is stored with the programme. In .NET or C# metadata can be added using *attributes*. Attributes can either be intrinsic or custom; examples of intrinsic attributes are: `[WebMethod]` or `[Serializable]`. Custom attributes are created by the developer, in AspectC# we use a custom attribute `[introduction(<string>)]` to enable the static crosscutting of methods and variables within base classes. In CLAW it is the metadata information that enables CLAW to identify methods before JIT compilation and then it may perform the aspect weaving. Metadata, in particular custom attributes is an enormous advantage the .NET framework gives the developer.

5.2.2 Reflection mechanism

Reflection is closely linked to metadata. Reflection is the process by which a programme can read its own metadata [Lib01]. In Java the reflection mechanism views data, has type discovery or *dynamic invocation* were properties and methods can be invoked at runtime, but with C# there is more powerful support for reflection. C# allows the ultimate goal of reflection, which is the creation of types at runtime this is called *reflection emit*. Many of the currently available AOP tools have their origins in reflection-based approaches more investigation is necessary to see if the improved reflection mechanism of the .NET framework would enable more reflection based AOP tools.

5.2.3 CodeDOM

The ability to model a source code file as a language neutral abstract syntax tree was of great benefit to our project. Some AOP approaches target source code weaving, such as AspectJ. By modelling source code as an AST there is the possibility of cross language aspect weaving. If a language can be modelled as a CodeDOM object graph then cross language aspect weaving would be relative easy as all the code is at the object graph level. But there needs to be more support from the CodeDOM API for expressions and statements to enable full AST modelling from any language. The CodeDOM API currently does not support the full Common Language Specification.

5.2.4 MSIL

Targeting the MSIL code for weaving aspects is used by most of the currently in development approaches to AOP within the .NET framework. The fact that C#, VB.NET and other supported languages are compiled into IL enables cross language weaving of aspects to base code. Also because the CLR JIT compiles IL before execution, this leaves the possibility of aspect weaving at runtime.

5.3 Evaluation Methodology

In the paper *Does Aspect-Oriented Programming Work?* [MWB$^+$01] Murphy et al. describe how to evaluate the usefulness of AOP. They did this by getting multiple organisations to implement products with and without AOP, then evaluate. They describe two basic techniques to evaluating programming technology: experiments and case studies.

Experiments are usually performed under controlled circumstances with a number of developers and test programs. Walker et al. [WBM99] performed experiments using AspectJ and three participants. The participants were asked to do two tasks: Debug an existing application and change an existing application. Case studies are of particular use in investigating the usefulness of an AOP language; examples include *Atlas* [MLWR01]. Larger case studies examine how the AOP technology affects multiple issues concerning the whole development of an application. Smaller case studies investigate specific development issues.

We decided that due to time restrictions that it would be only possible to perform a small case study, but this would help with evaluating specific issues relating to development with AspectC#.

5.4 AspectC# versus The Rest

We decided to evaluate AspectC# against four "state of the art" tools: AspectJ, Hyper/J, CLAW and AOP#. It was decided that evaluation would be performed under six issues (issues 1 - 5 were taken from Elrads article *Discussing Aspects of AOP* [EFB01], issue 6 was added into the evaluation because the usability of a programming tool is very important.):

1. How the AOP system specifies aspects.
2. What composition mechanisms the system provides.
3. Implementation mechanisms.
4. Decoupling.
5. Software process.
6. Usability.

Before the evaluation begins, it must be stated that the Java solutions have been in development for a longer period of time and show the greater progression in AOSD. Most of the .NET solutions borrow ideas from the Java based AOP tools. But the .NET approaches bring experience with cross language aspect weaving and aspectual polymorphism which are very promising ideas in AOSD.

5.4.1 How the AOP system specifies aspects

Table 5.1 evaluates how each approach specifies aspects; this issue relates to:

1. How the tool defines join points (places in the program were base or core code interacts with the aspect code).
2. What is the actual source code requirement, can aspects be applied to variable, method or class level? Probably not too useful if just class level.
3. The extent to which an aspect may be customised to a certain usage within the system. Can aspects be general or specific? A system which allows both would be of greater value to the developer.

5.4.2 Composition mechanisms the system provides

Table 5.2 show each composition mechanism for each approach, this issue relates to:

1. Whether there exists a dominant decomposition within the language. Is one decomposition applied to aspects or are all concerns treated as equals.
2. Does the system contain explicit language extensions for aspects? If the tool makes explicit language extension then it forces the developer to learn these extensions and makes source code that does not conform to the language specification.
3. What is the relationship, visibility between aspects? In order for reusability of aspects the relationship between aspects must be fully described.
4. Can aspects influence behaviour and is there a mechanism for resolving conflicts between aspects?

Tool	How AOP system specifies aspects
AspectJ	AspectJ consists of a Join Point model, were: (i) A join point is a well-defined point of execution in the program, as discussed in table 2.1. (ii) Pointcuts are a set of join points. An aspect is a modular unit of a crosscutting concern. Aspects can be customised to specific requirements or be general and effect multiple areas of the system.
Hyper/J	In Hyper/J join points are defined by relationships between concerns. Joining occurs at method level. Each concern or artefact within a dimension is encapsulated. Concerns are specific within the dimension.
CLAW	CLAW uses a similar syntax to AspectJ; it employs the same join point model but it is a subset of AspectJ with join points only at the method level. The mapping of base to aspect code is achieved through a XML weave definition file. An aspect in CLAW has the same meaning as in AspectJ. Aspects can be customised to specific usages or general.
AOP#	AOP# uses a similar syntax to AspectJ. A subset of the join point model of AspectJ is used. The mapping of core to aspect code is achieved through an XML connector file. An aspect in AOP# has the same meaning as in AspectJ. Aspects can be customised to specific usages or general.
AspectC#	AspectC# uses a similar syntax to AspectJ. A join point is a well-defined point of execution: a method execution. In AspectC# an aspect is a modular unit of crosscutting implementation. Aspects can be designed to be either general or specific.

Table 5.1: How AOP system specifies aspects

5.4.3 Implementation mechanisms

Table 5.3 discusses the implementation mechanisms of each tool. This issue includes all aspects of the implementation from:

1. When the composition is determined. Static is when compositions are determined at compile time and dynamic is when compositions are determined at run time.

2. Is there a clean separation of aspect and base code, can the compilation oc-

Tool	Composition mechanisms the system provides
AspectJ	The dominant decomposition within Java is OO, as AspectJ is applied to Java all aspects are applied to class structures. AspectJ extends Java with explicit language extensions. Aspects within AspectJ can influence the behaviour of the main program namely with the `proceed` keyword. AspectJ includes rules for resolving conflicts among aspects.
Hyper/J	Hyper/J stems from SOP and is an implementation of Multi-Dimensional Separation of Concerns (MDSOC) that supports arbitrary concerns in the form of a hyperslice; therefore there is no dominant decomposition and different decompositions of a system may coexist. Hyper/J does not have any language extensions. It is possible for hyperslices to interact between dimensions and for hyperslices to influence the behaviour of a system.
CLAW	The CLR currently only supports OO languages, this means that CLAW affects only OO languages although there is the possibility of support for other decomposition mechanisms. CLAW does not impose any language extensions.
AOP#	AOP# also works within the CLR; therefore AOP# affects only OO languages although there is the possibility of support for other decomposition mechanisms. AOP# does not impose any language extensions. AOP# uses the concept of *'aspectual polymorphism'* were aspects can be turned on and off dynamically at runtime.
AspectC#	AspectC# does not extend the C# language. The dominant decomposition is OO as aspects are applied to classes. AspectC# does not make any language extensions to C#. AspectC# has similar constructs to AspectJ and can dynamic alter the behaviour of methods using the `proceed` keyword.

Table 5.2: Composition mechanisms the system provides

cur separately? Does the tool make invasive changes to the base leaving the actual base code file changed after the tool.

3. What is the target code of the system? If the target source code then the developer must have access to the base source files, if the target is byte code or an intermediate language then the developer may only need the byte code or intermediate class file.

4. Can the system be applied to existing applications?

5. Is there a way of verifying compositions? Is it possible to tell if the aspect to base composition is correct?

6. Does system contain support for Aspectual Polymorphism? Can aspects be turned on and off?

Tool	Implementation mechanisms
AspectJ	In AspectJ compositions are determined statically at compile time. There is a clear separation of aspect and base code; but it is possible to compile and run base code separately.AspectJ may work on new or existing systems. AspectJ targets source code and it possible to verify compositions using the compiler. No support for Aspectual Polymorphism.
Hyper/J	Hyper/J performs compositions are compile time. Hyperslices can be compiled separately or as a whole system. Hyper/J can be applied to new or existing systems. Hyper/J targets byte code and it is not possible to verify compositions. No support for Aspectual Polymorphism.
CLAW	CLAW performs composition dynamically at runtime. There is a clear separation of aspect and base code. CLAW can work on new or existing systems. CLAW targets MSIL before JIT compilation and there is no composition verification mechanism. No support for Aspectual Polymorphism.
AOP#	AOP# performs composition dynamically at runtime. There is a clear separation of aspect and core code. AOP# can work on new or existing systems. AOP# targets MSIL, and currently no support for verifying compositions. Supports for Aspectual Polymorphism.
AspectC#	Compositions in AspectC# are performed statically at compile time. There is a clear separation of aspect and base code. AspectC# can be applied to new or existing systems. AspectC# targets source code, currently it is not possible to verify compositions.

Table 5.3: Implementation mechansims

5.4.4 Decoupling

Decoupling relates to whether the developer of the base code is aware of the aspect code and does the developer have to make adjustments to prepare for aspect code? This is called obliviousness [EAK$^+$01], the base code developer is oblivious of the aspect code. Following from this is it possible for the developer to apply aspects globally, locally or both? It would be better if the developer was left the option of applying aspect either globally or locally. Table 5.4 describes each approach to decoupling in detail.

Tool	Decoupling
AspectJ	AspectJ leaves the option of obliviousness to the developer. The developer may have to prepare for aspects as in the case of transactions. Aspects can affect the program locally or globally.
Hyper/J	Concerns in Hyper/J are self contained and do not know about other concerns. The developer must have intimate knowledge of the system, but if Hyper/J is applied to an existing application. Concerns can affect the program locally or globally.
CLAW	CLAW has a similar syntax to AspectJ, the option of obliviousness is left to the developer. Aspects can affect the program locally or globally.
AOP#	AOP# has a similar syntax to AspectJ, the option of obliviousness is left to the developer. Aspects can affect the program locally or globally. There is also the ability to switch on and off aspects.
AspectC#	The option of obliviousness is left to the developer. In some cases it is necessary for the base code to 'prepare' for aspect advice, while in other cases it is not necessary. Aspects can be either global or local.

Table 5.4: Decoupling

5.4.5 Software Process

This issue includes the overall software process:

1. What methodology or framework the system uses?

2. What aspect mechanisms enable reuse?

3. Is it possible to analyse the performance of the aspect system.

4. Is there a debugging mechanism?

Each software process is discussed in Table 5.5.

Tool	Software Process
AspectJ	AspectJ uses AOP with a clear separation of base code and aspect code. An aspect in AspectJ is a modular unit of crosscutting concern; this structure enables reuse. Aspects can be either general or specific. Currently there is no debugger for AspectJ.
Hyper/J	Hyper/J uses MDSOC methodology that can deal with arbitrary concerns. A hyperslice enables reuse within Hyper/J. Concerns usually are general but can be specific. Hyper/J is an on going project and currently there is no debugger.
CLAW	CLAW uses AOP with a clear separation of base code and aspect code. An aspect in CLAW enables reuse. Aspects can be either general or specific. Currently there is no debugger for CLAW.
AOP#	AOP# uses AOP with a clear separation of base code and aspect code. An aspect in AOP# enables reuse. Aspects can be either general or specific. Currently there is no debugger for AOP#.
AspectC#	AspectC# uses AOP with a clear separation base and aspect code. An aspect is a modular unit of a crosscutting concern and this structure enables reuse. Currently there is no debugger for AspectC#.

Table 5.5: Software Process

5.4.6 Usability

The usability issue relates to the up take of the language, how easy it is to programme with the system. Is there any developer support or community mailing lists? Table 5.6 discusses the usability of each tool.

Tool	Usability
AspectJ	AspectJ is user friendly with IDE developer plug-ins also available. There is a lot of research into the development with AspectJ and there exist a large user community group.
Hyper/J	Hyper/J is more a complex approach but does have GUI support. There is also a medium size community group.
CLAW	Currently no prototype available.
AOP#	Currently no prototype available. But because AOP# uses similar semantics and constructs to existing AOP tools it is hoped that it will be usable from the start.
AspectC#	The alpha version of AspectC# has just been released. We hope to gain a lot of interest and feedback from its release. The fact that it uses similar syntax to AspectJ we hope will enable greater usability.

Table 5.6: Usability

5.5 An Example: A Bank Account

In this example we demonstrate the use of AspectC# from design to implementation. We will show an example of development with AspectC#.

5.5.1 The Problem

The problem is a simple Bank account scenario: when a user deposits or withdraws money from their account we would like the entire task to work or none of it to work, i.e. transactions. Transactions are based upon the ACID principle [Cou01]. Transactions are a crosscutting concern as they result in non-modular code.

5.5.2 OO Solution

The design is simple and consists of four classes: A super class that represents a bank account and three sub classes that represent types of accounts: current, savings and business accounts. Figure 5.1 shows a class diagram of the solution.

The solution consists of four classes:

1. *Account*: Super class for an account, holds details about a individual client.

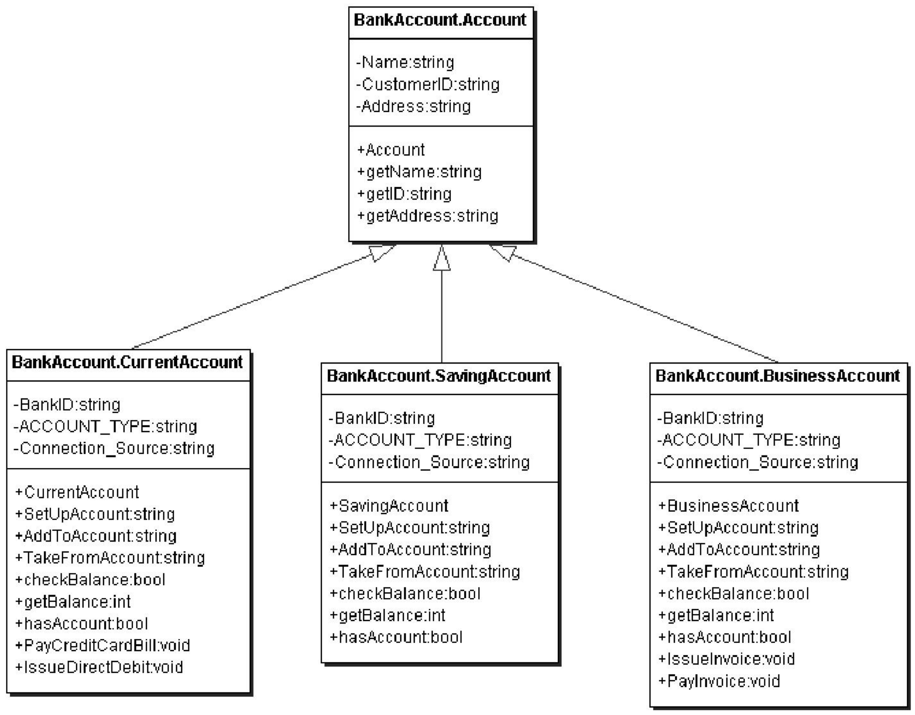

Figure 5.1: Bank account class diagram.

2. *CurrentAccount*: Represents a current account for the client and has the functions of deposit, withdrawal, pay bill, etc.

3. *SavingsAccount*: Represents a savings account for the client and has the functions of deposit, withdrawal, calculate interest, etc.

4. *BusinessAccount*: Represents a business account for the client and has the functions of deposit, withdrawal, pay bill, invoices, etc.

The following code segment is taken from `CurrentAccount`, it represents the procedure for deposits into the account. Transaction start with a *begin transaction* command and either end successfully with a *commit* command or unsuccessfully with a *rollback* command.

```
SqlConnection connection = new SqlConnection(
Connection_Source);
```

```
3. connection.Open();
4. SqlTransaction tx = connection.BeginTransaction();
5. string message = "";
6.
7.  try
8.   {
9.      string CommandText = "Update Account " +
10.        " Set Balance = Balance + " + Amount +
11.        " Where CustomerID = ’" + base.getID() + "’" +
12.        " And Type = ’" + ACCOUNT_TYPE + "’";
13.     SqlCommand command = new SqlCommand();
14.     command.CommandText = CommandText;
15.     command.Connection = connection;
16.     command.Transaction = tx;
17.     command.ExecuteNonQuery();
18.
19.     tx.Commit();
20.     message = "Update suceeded!";
21.  }
22.  catch(Exception e)
23.  {
24.     tx.Rollback();
25.     message = "Record not written to the database!";
26.
27.  }
28.  finally
29.  {
30.    connection.Close();
31.  }
```

From the code segments lines 1-4 deal explicitly with the opening of the connection and starting of the transaction. Lines 16 and 19 deal with associating the transaction with the SQL update and committing the transaction if it was successful. Finally Lines 22 - 31 deal with the situation if an error occurred and the closing of the transaction. If we examine the updating of the record to the database only lines 9 - 14 deal explicitly with the updating of the record the rest are needed for opening the connection to the database and the transaction code. In the OO solution similar code in each class is necessary for the application to handle transactions.

5.5.3 AspectC# Solution

The AspectC# solution is similar to the OO solution as shown in figure 5.1, but the transactions has been taken out from these classes and put into an `Aspect` class called `AspectTransaction`. In this class all the connection and transaction management is handled. `AspectTransaction` consists of two aspect methods:

1. `beforeTransaction`: Handles connection and transaction setup.
2. `afterTransaction`: Handles commit\ rollback of transaction and closing of connection.

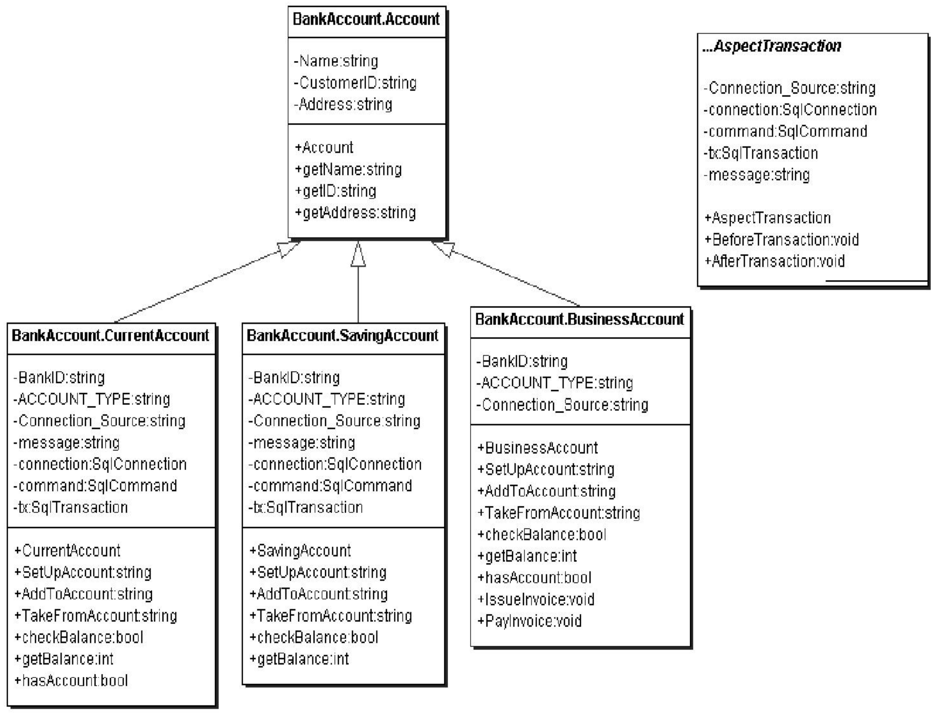

Figure 5.2: Bank account AOP class diagram.

The following code segment shows the two methods beforeTransaction and afterTransaction from AspectTransaction.

```
public void BeforeTransaction()
```

```
{
   connection = new SqlConnection(Connection_Source);
   connection.Open();
   tx = connection.BeginTransaction();
   command.Transaction = tx;
}

public void AfterTransaction()
{
   if(message.Equals("Error"))
   {
      tx.Rollback();
   }
   else
   {
      tx.Commit();
   }

   connection.Close();
}
```

The next code segment shows how the base class method is affected by the aspect class. We can see that only the necessary SQL is contained within the method therefore making the code easier to understand and more modularised. But in the base code it is necessary to move the connection variable out of the method and prepare of the aspect code. We do not see this as too much of a limitation as the resulting code after applying AspectC# is well modularised.

```
   try
   {
      string CommandText = "Update Account " +
             " Set Balance = Balance + " + Amount +
             " Where CustomerID = ’" + base.getID() + "’" +
             " And Type = ’" + ACCOUNT_TYPE + "’";

      command.CommandText = CommandText;
      command.Connection = connection;
      command.ExecuteNonQuery();
      message = "Balance updated successfully";
   }
```

```
catch(Exception e)
{
   Console.WriteLine(e.StackTrace);
   message = "Error";
}

return message;
```

5.5.4 Results

We will base the evaluation of the result of the development with AspectC# on three criteria:

1. Number of Lines of code versus OOP solution: the less number of lines of code means the less the developer has to write and also for maintenance the less the better.

2. How modular are the crosscutting concerns after AspectC#: By modularising crosscutting concerns we do not have problems like code scattering and tangling, this also aids development and maintenance.

3. Usability of AspectC#: AspectC# is a programming tool, the less usable it is the less likelihood it will be useful to developers.

Lines of Code

From figure 5.3 we can see in the OO solution that on average each class consists of approximately 200 - 250 lines of code in the OO solution. The transaction code is highlighted in each class, we can see that it is scattered and tangled within the class structure.

From figure 5.4 we can see in the AOP solution that the average class contains approximately 170 - 200 lines of code in the AOP solution but there is the addition of another class the aspect class. But what we find is that if more transaction are

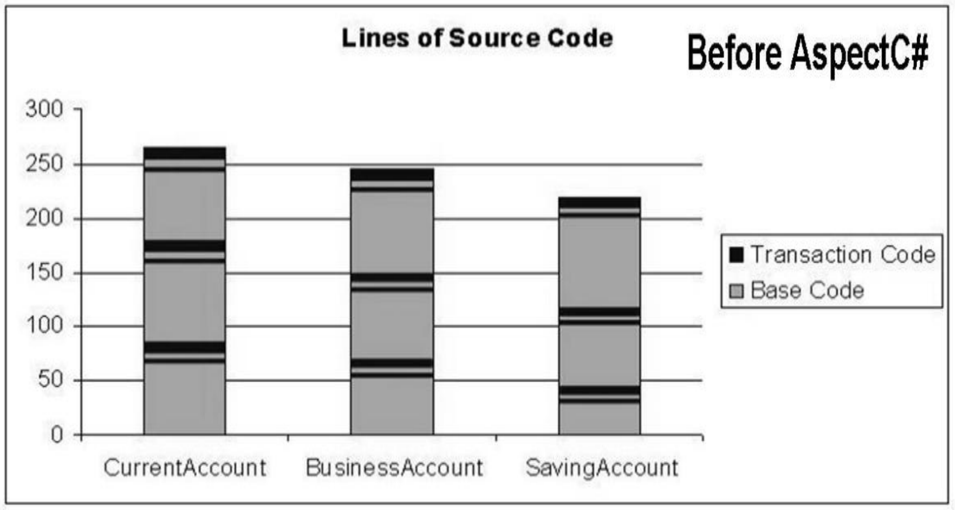

Figure 5.3: Number of lines of code from OO solution.

necessary then there is no need to tangle code within the base classes all that is needed is an additional entry in the Aspect Deployment Descriptor.

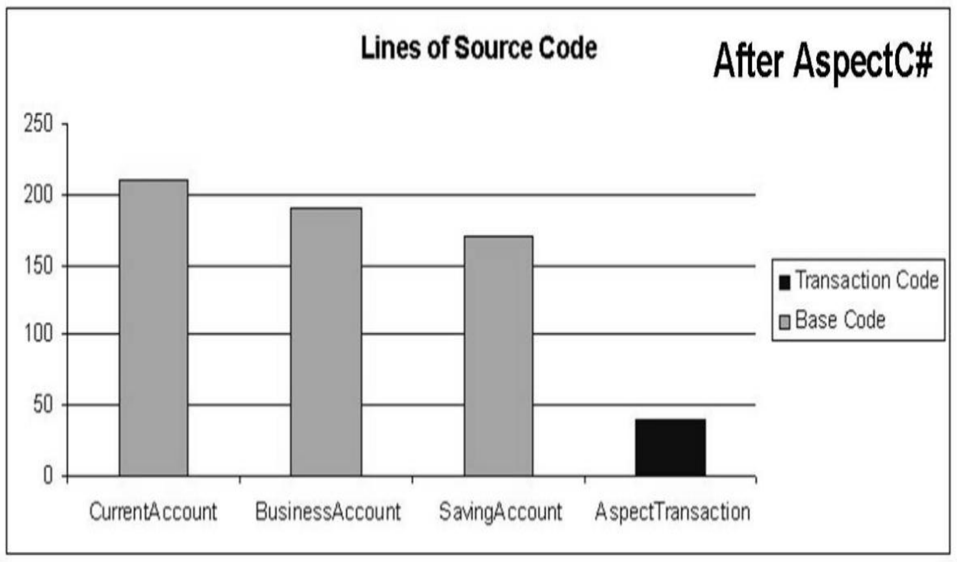

Figure 5.4: Number of lines of code from AOP solution.

Modularisation of crosscutting concerns

From figure 5.4 we can see how the tangled and scattered code associated with transactions is taken out from the base classes and modularised into a single aspect class. This results in the code being easier to maintain and lowering the likelihood of errors being introduced when changes to the application are necessary.

Usability of AspectC#

It is quite difficult to evaluate the usability of AspectC#, it is only a research prototype and we would need a large user community to properly evaluate the usability of the tool. But what we can say is that AspectC# is based upon AspectJ and its semantics are user friendly. The development environment of AspectC# is the Visual Studio .NET IDE, which is extremely user friendly and is the de facto IDE for .NET development.

5.6 Summary

In this chapter we have evaluated .NET as an AOSD platform and our tool AspectC#. Firstly we discussed what a platform is and how .NET fits into the AOSD scheme. Then we compared our tool with other leading AOP tools from both Java based solutions to .NET based solutions. We have also covered a small example of using AspectC# in a development environment comparing it against OOP with C#.

Chapter 6

Conclusions

6.1 Introduction

In this chapter we discuss the conclusions of the project. Firstly we begin with a discussion of the objectives of the dissertation and how well these objectives were met. Next there is an outline of the benefits of AspectC# over the competition. Then the limitations of the system are considered and finally the future development of AspectC# is discussed.

6.2 Summary of Objectives

In chapter 1 we discussed the goals or objectives of the dissertation. We believe that we have meet, if not exceeded each goal of the dissertation. We will now look at each objective in turn:

1. Design and Implementation of AspectC#

   Chapters 3 and 4 dealt with the design and implementation of AspectC#. We believe we have created a tool that meets all requirements set our in the requirements of out system. The modular design of the system has left the possibility of extending the system to enable cross language weaving. A thorough discussion on the implementation issues was also described.

2. Evaluation of AspectC#

In chapter 5 we discussed the evaluation of AspectC#. This evaluation covered but a comparison against currently available tools \ methodologies and a simple example using AspectC#. We showed how AspectC# reduces the number of line of code for crosscutting concerns and how modular the code becomes after running AspectC#.

3. Investigation of AOP technologies

   In chapter 2 we investigated the state of the art of AOSD tool support. A full evaluation of each AOP approach was carried out this included: composition mechanisms, implementation mechanisms, architecture and semantics. In summary:

   - AspectJ is probably the leading AOP tool currently available. AspectJ has a large user group and a lot of tool support with IDE plug-ins. AspectJ can best be described as a source pre compiler that weaves aspects at compile time. AspectJ is currently on release 1.0 and its semantics are quite stable at present. Alot of other tools base their semantics around AspectJ including our own tool AspectC#.
   - Hyper/J is more complex than AspectJ. The Hyper/J community is growing steadily. Hyper/J is the only tool to base their semantics on Multi Dimensional Separation of Concerns (MDSOC).
   - CLAW is still not released and until its security problems are handled it will not be released. But it promises to be a useful tool from the beginning, the cross language weaving feature is a major advantage and the fact it targets the entire .NET platform it is sure to attract more users into the AOSD community.
   - AOP# is soon to be released and is quite similar to CLAW in its approach to weaving and AspectJ in its approach to the semantic constructs. The aspectual polymorphism is something that no tool currently has.

4. AOSD within the .NET Platform

   The .NET framework was developed to be the successor to Windows platform, among one of its many goals is easier development of applications. But does the .NET platform offer anything more than the currently available platforms to AOSD? In Chapter 5 we discussed areas of the .NET platform that we think can offer or support added AOSD techniques.

6.3 Benefits

We believe we have created a tool that enables the modularisation of crosscutting concerns within C# without explicit language extensions and is easy to use. The system was designed to enable extensibility and hopefully with full CodeDOM support cross language aspect weaving will be available in later versions of the system.

To our knowledge AspectC# is one of the firstly available tool that enables AOP mechanisms within the .NET platform (Both AOP# and CLAW are soon to be released). Hopefully this tool will enable the .NET community to get more involved with AOSD, thereby increasing the understanding of AOSD.

6.4 Limitations of System

AspectC# is only a research prototype and there exist many limitations to the system. We have identified areas of the system that we think limit the system in the performance of applying AOSD within C#. The areas are covered in the following subsections.

6.4.1 Parser Limitations

Currently the implemented parser cannot handle:

- Inner classes.

- structs.
- Delegates and Events.
- Attributes.
- Expressions and statements within methods.

6.4.2 CodeDOM Limitations

CodeDOM Limitations The CodeDOM API is part of the .NET base class library. Currently, the elements are not supported by CodeDOM API are:

- Variable lists e.g. `int i,j,k;`
- Parameters keyword: `params`
- Lack of support for expressions and statements
- Attribute targets

6.4.3 Semantic Limitations

AspectC# semantics are a subset of the AspectJ semantics, the semantics of AspectJ that are not supported by AspectC# are:

- Join Point Model: AspectC# only supports join points at method execution. There needs to be full support at method call, exception handling and static initialisation within classes.
- AspectC# cannot support the cflow keyword from AspectJ.

6.5 Future Work

We believe that AspectC# has achieved it main objective of modularisation of crosscutting concerns. But for AspectC# to become the leading .NET based AOP tool we have identified five areas that we think merit future research.

6.5.1 Parser

The parser that was implemented for the project was a simple and quickly developed parser. For the progression of AspectC# there needs to be a better parser. Of the currently available parsing tools the most promising are:

- C# grammar for Flex \ Bison [BIS]: This a script that enables flex \ bison to parse C# files.

- jb2csharp: This is port of JB Parser and Lexer Generation for Java (currently in alpha version).

- Mono project: The mono project are implementing an open source .NET platform, which includes a C# parser.

6.5.2 CodeDOM Graph

The CodeDOM API is of great advantage to us, but there needs to be more support for modelling code constructs especially within methods and particularly for expressions and statements.

6.5.3 Semantics of AspectC#

AspectC# uses the same semantics as AspectJ, this is fine for the prototype version. But there needs to be more investigation into the development of other semantics for AspectC# and especially the .NET framework.

6.5.4 Testing

The only true way of evaluating a programming technique is to develop systems with that technique. Clearly one small case study is not enough, there needs to be more case studies and experiments with AspectC# to fully evaluate the usefulness of the tool.

6.5.5 Cross language weaving

If there exists full parsing support for all languages and full CodeDOM support this will enable cross language aspect weaving at the source code level.

Bibliography

[AOP] AOP#. Unreleased Egon.Wuchner@mchp.siemens.de.

[AOS] AOSD. AOSD Homepage. http://www.aosd.net.

[ASPa] AspectC++ Homepage. http://www.aspectc.org.

[ASPb] AspectR Homepage. http://aspectr.sourceforge.net/.

[BIS] Bison. http://dinosaur.compilertools.net/#bison.

[Boo98] G. Booch. *The Unified Modelling Language User Guide*. Addison Wesley, 1st edition, 1998.

[Cla] Siobhan Clarke. Theme UML Homepage. http://www.dsg.cs.tcd.ie/s̃clarke/ThemeUML/.

[Coa] Yvonne Coady. AspectC Homepage. http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[COM] Compoisition Filters Homepage. http://trese.cs.utwente.nl/composition filters/.

[Cou01] George Coulouris. *Distributed Systemsm, Concepts And Design*. Addison-Wesley, 3rd edition, 2001.

[DEC] DECHOW Homepage. http://cs.oregonstate.edu/d̃echow/.

[Dij76] E.W Dijkstra. *A Discipline of Programming*. Prentice Hall, 1st edition, 1976.

[EAK$^+$01] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications Of the ACM*, 44(10):33–38, 2001.

[EFB01] Tzilla Elrad, Robert Filman, and Atef Bader. Aspect-oriented programming. *Communications Of the ACM*, 44(10):28–32, 2001.

[Hir] Robert Hirshfled. AspectS Homepage. `http://www.prakinf.tu-ilmenau.de/h̃irsch/Projects/Squeak/AspectS/`.

[HO93] W. Harrison and H. Ossher. Subject oriented programming (a critique of pure objects). *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1993.

[HYP] HyperJ Homepage. `http://www.alphaworks.ibm.com/tech/hyperj`.

[is.] What is.com. Whatis.com. `http://www.whatis.com`.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. *Proceedings of the 5th European Conference on Object Oriented Programming (ECOOP)*, 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. *Proceedings of the European Conference on Object Oriented Programming (ECOOP)*, 1997.

[Lam] John Lam. CLAW Homepage. `http://www.iunknown.com`.

[LEX] LEX Homepage. `http://dinosaur.compilertools.net/`.

[Lib01] Jesse Liberty. *Programming C#*. O'Reilly, 1st edition, 2001.

[Lie] Karl Lieberherr. DemeterJ Homepage. `http://www.ccs.neu.edu/research/demeter/DemeterJava`.

[Mica] Microsoft. Aspect.Net project description. `http://research.microsoft.com/programs/europe/rotor/Projects.asp`.

[Micb] Microsoft. C# Specification Homepage. `http://msdn.microsoft.com/net/ecma/`.

[Micc] Microsoft. CODEDOM Homepage. `http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemcodedom.asp` .

[Micd] Microsoft. Microsoft under the hood column. `http://msdn.microsoft.com/msdnmag/columns/hood.asp`.

[MLWR01] Gail Murphy, Albert Lai, Robert Walker, and Martin Robillard. Separating features in source code: An exploratory study. *Proceedings of the 23st International Conference on Software Engineering*, 2001.

[MWB$^+$01] Gail Murphy, Robert Walker, Elisa Baniassad, Martin Robillard, Albert Lai, and Mik Kersten. Does aspect-oriented programming work? *Communications Of the ACM*, 44(10):75–78, 2001.

[OT01] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.

[Par72] D.L Parnas. On criteria to be used in decomposing systems into modules. *Communications of the ACM*, 1972.

[Per] Perldoc.com. Perl5 Regular Expressions. `http://www.perldoc.com/perl5.6/`.

[SAN] Visual Parse Homepage. `http://www.sand-stone.com/`.

[Tea] AspectJ Team. AspectJ Homepage. `http://aspectj.org`.

[TO00] Peri Tarr and Harold Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, first edition edition, 2000.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[WBM99] Robert Walker, Elisa Baniassad, and Gail Murphy. An initial assessment of aspect-oriented programming. *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[YAC] YACC Homepage. `http://dinosaur.compilertools.net/`.