Aspect-Based Properties

A thesis submitted to the
University of Dublin, Trinity College,
in fulfilment of the requirements for the degree of
Doctor of Philosophy (Computer Science).

Donal Lafferty

Distributed Systems Group,
Department of Computer Science,
Trinity College, University of Dublin.

October 2004.

DECLARATION

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this or any other University, and that, unless otherwise stated, it is entirely my own work.

Donal Lafferty,
22th October 2004.

PERMISSION TO LEND AND/OR COPY

ACKNOWLEDGEMENTS

Front and centre:
Special thanks to Prof Vinny Cahill for forcing me to see the big picture, but also for giving me the time and breathing space to draw a box around an idea and to fill it in.

Behind me all the way:
My dearest love to Irenee Daly for her patience and consideration during the writing of this thesis and for seeing to a sense of self-confidence for myself.

The Family Crew:
Mom and dad, thanks for taking a big chance on this research thing. The investment and patience really made it possible, but let me not forget Kate, who provided me with sensibility in the later stages.

The Lads and Ladies:
In fairness, researching with the Distributed Systems Group is a life style choice; however, I would like to point out some people who really made an impression during my stay: Tim Walsh, Raymond Cunningham, Jean-Marc Seigneur, Stefan Weber, Jim Dowling, Johan Andersson

Of course, the atmosphere would not have been as much fun without these characters old and new: Sean "Hit me baby one more time" Blanchfield, Tilman "Dr. T" Schaefer, Mads "Madsarati" Haahr, John "JFK" Keeney, Andronikos "Open Source" Nedos, Sergey "The Smoking Man" Tsvetkov, Simon "Pub Crawl!" Dobson, Kulpreet "Mr TB" Singh, Anthony "Kildare" Harrington, Liz "The Republican" Gray, Gregor "The German" Gaertner, Elizabeth "Full of Personality" Drew, Rene "Be my Valentine" Meier, Barbara "Marathon" Hughes, Peter "Crazy Hair" Barron, Vinny "Street Figher" Reynolds, Greg "Quiet Man" Biegel, Elisa "Hilarious" Baniassad, Joerg "Dancing Man" Abendroth, Bartosz "The Overlord" Biskupski, Dave "Sideshow Bob" Gregg, Cormac "Free Concert Tickets" Driver, Andrew "Vodka Bar" Jackson, Shiu "Chewy" Lun Tsang, Daire "The Medatator" O'Broin, Domink "The Running Man" Dahlem, Ivana "Bosnians rule!" Dusparic, Elizabeth Daly, Melanie Bouroche, Lotte "Shutup Tim!" Nickel, Jan "Unforgiven" Sacha, David "Smiling Dave" McKitterick, Andrew "Remote-" Edmonds, Colin "Tall Man" Harris, Stephen "Shinner" Farrell, Ryan "Who?" van Roode, Rita McGuinness, Chen "No Proxies" Yong, Melanie "Microwave" Bouroche, Aline "En Francais" Senart, Malika "Bad Education" Boulkenafed, Daniel "Mr. Carribean" Molinard, Alan "TinTin" Gray, Kari "I'm Married" Schougaard, Marco “Dancing Man” Killijan.

Thanks everyone, I could not have done this on my own.
–Donal Lafferty

Summary

Component frameworks are said to support contextual composition when crosscutting functionality is bound to component instances by declarative selection of context properties, rather than through direct connections, such as method invocation, or derivation mechanisms, such as inheritance. Using contextual composition, component framework services such as synchronization, security and transaction support are bound to component instances via method interception. Here, the term component instance is an abstraction for whatever unit of interaction is used to access software component functionality be it an interface, an object or a set of objects. The mechanism for declarative selection can range from deployment descriptors, used with EJB Containers, to attribute-based annotations to source, used with CLR contexts of the .NET Framework.

Contextual composition frameworks suffer from the *lack of tailorability problem* as well as the *preplanning problem.* Contextual composition is employed in a range of component frameworks including MTS contexts, EJB containers, COM+ contexts, CCM containers, and CLR contexts. The lack of tailorability problem arises because the context properties available are either fixed or extensible in an ad hoc manner. The preplanning problem arises because accessing context properties constrains component architecture. Binding to context properties involves exposing component functionality as instance methods and supplying significant prerequisite composition infrastructure.

Aspect-oriented programming (AOP) addresses the problems of contextual composition, but AOP solutions are difficult to adopt as they introduce language dependencies and suffer problems with reusability. AOP offers language extensions that provide a linguistic means of implementing new crosscutting concerns encapsulated in aspects. An emphasis on noninvasive binding means AOP places fewer restrictions on component architecture, but relying on language extensions forces components to align to a single language for interoperability. Furthermore, reusability involves the customization of an aspect, which is much more complex than declarative mechanisms used with contextual composition such as attribute-based property selection.

This thesis introduces aspect-based properties, which avoid the restrictions of context properties, provide language-independence and simplify reuse. Aspect-based properties are implemented by aspects with pointcut-advice semantics, and composition is the responsibility of the aspect weaver rather than the components being composed. The underlying aspect model is language-independent in that it allows aspects and components to be written in a variety of languages and freely intermixed. Aspect-based properties use attribute-based property selection to allow reuse without the need to customise an aspect.

An implementation for standardised Common Language Infrastructure (CLI) demonstrates aspect-based properties to be easy to adopt and to solve the problems identified with context properties. Aspect-based properties are implemented as CLI components with XML-based crosscutting specifications that are composed with application components using a load-time weaver. For reusability, aspect-component bindings are written in terms of attributes types, but for support of legacy components custom crosscutting is available in which bindings are specified in terms of CLI metadata. Language-independence is available in either case, which we demonstrate by weaving aspect-based properties and components written in object-oriented, procedural and functional programming languages. In comparison to the CLR contexts for the CLI, aspect-based properties provide a richer join point model for better tailorability, the weaver allows them to avoid preplanning issues, and they execute an order of magnitude faster.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

“Get to the point.”
–Attributed to Tim Walsh

Component frameworks are said to support contextual composition when crosscutting functionality is bound to component instances by declarative selection of context properties, rather than through direct connections, such as method invocation, or derivation mechanisms, such as inheritance [Szy'02]. Using contextual composition, component framework services such as synchronization, security and transaction support are bound to component instances via method interception. The term *component instance* [Szy'02] is not rigorous, and in practice a component instance can range from an interface of static functions to an object to a set of objects. The commonality is that these elements form the basis of the API from which component functionality is accessed. The mechanism for declarative selection can range from deployment descriptors, used with EJB Containers [DeM'03], to attribute-based annotations to source, used with CLR contexts that are available with the .NET Framework [Mic'04b] implementation of the Common Language Infrastructure (CLI) specification [Ecm'03b].

Contextual composition frameworks suffer from the *lack of tailorability problem* [Pic'03] as well as the *preplanning problem* [Tar'99, Cli'00]. Contextual composition is employed in a range of component frameworks including MTS contexts [Gra'97], EJB containers [DeM'03], COM+ contexts [Edd'99], CCM containers [OMG], and CLR contexts [Mic'04b]. Such component frameworks are sometimes referred to as container models [Pic'03, Coh'04]. The lack of tailorability problem, or tailorability problem, arises because the context properties available are either fixed or extensible in an ad hoc manner. In many cases, the functionality available as context properties is fixed as in the case of MTS context [Szy'02], COM+ contexts [Szy'02], CCM containers [Duc'02] and standard EJB containers [Szy'02]. Extensibility is available with CLR contexts [Shu'02, Szy'02] and non-standard extensions to EJB [jBo'01, Pic'03]; however, these programming models are idiom-based, and this makes them difficult to use [Pic'03]. The preplanning problem arises because accessing context properties constrains component architecture. Binding to context properties involves exposing component functionality as instance methods and supplying significant prerequisite composition infrastructure. For example, EJB [DeM'03] requires the specification of several interfaces to describe which instance methods will be exposed by a container, and CLR contexts place inheritance restrictions on types bound to context properties [Shu'02, Szy'02].

This thesis introduces *aspect-based properties,* which use AOP to avoid the restrictions of context properties, but avoid barriers to adoption by providing language-independence and simplifying reuse. Aspect-based properties are implemented by aspects with pointcut-advice semantics [Kic'01b, Kic'01a, Mas'03], which provide a well-structured programming model for writing custom properties [Coh'04] and are shown in this thesis not to interfere with component architecture. Aspect-based properties avoid making component interoperability language dependent by adopting a language-independent AOP model [Laf'03], in which aspect-based properties and the components to which they are applied can be written in a variety of languages and freely intermixed. Language-independent AOP meets adoption criteria that requires AOP solutions not make existing component programming technologies [Coh'04] obsolete, and language-independent AOP is consistent with the language-independent nature of component-oriented programming [Szy'02]. Aspect-based properties address reusability by supporting attribute-based property selection [Shu'02, Szy'02]. We argue that the use of attribute types for property selection is consistent with the noninvasive properties of AOP by avoiding the need to modify component implementation, and we show that doing so allows reuse without the need to write new aspect code. Avoiding the need to learn how to revise crosscutting semantics also meets adoptability criteria, because it allows the component programmer to access aspect-based properties with existing knowledge.

The claims above are verified with an aspect weaver implemented for the CLI. This weaver allows the language-independent properties of aspect-based properties to be verified, which involves showing that aspects and components can be developed in a variety of languages and freely intermixed and showing that attribute-based property selection can be used regardless of component implementation language. Attribute-based property selection demonstrates the ability to reuse aspects without having to modify their crosscutting specifications. The weaver is also used to compare aspect-based properties with equivalent functionality written as context properties using CLR contexts, as CLR contexts are also implemented for the CLI. In this comparison, we note that the pointcut-advice mechanism offered by aspect-based properties provides better tailorability than CLR contexts, because the mechanism's join point model offers a richer set of execution points at which it can influence application semantics. Furthermore, the aspect weaver avoids the architecture restrictions that cause the preplanning problem associated with context properties. Finally, the execution overhead of aspect-based properties is an order of magnitude lower than that of CLR contexts.

In the remainder of this chapter, we focus on the contribution that aspect-based properties make to the area of composing software components and crosscutting concerns. The following section starts with an examination of the problem of composing crosscutting functionality with software components, and then explains how context properties provide only a limited solution to the difficulties faced. The next section outlines our strategy for improving on contextual composition. Next, we describe how aspect-based properties provide a novel solution to these requirements. Since this thesis limits its scope to introducing a new aspect-oriented programming technology and comparing it to contextual composition, we take another section to point out what falls outside the scope of our investigation. Following this, we outline the evaluation criteria that must be met to justify the value of aspect-based properties. We conclude this chapter with an overview of the remainder of the thesis.

1.1 Crosscutting using Contextual Composition

Typically, crosscutting functionality introduces tangling [Lop'97], which we can demonstrate to adversely affect the independent deployment and third-party composition characteristics of software components. Contextual composition solves these problems by avoiding the need to bind crosscutting functionality with tangling. However, the applicability of contextual composition is limited by the lack of tailorability problem and the preplanning problem. In the following subsections, we expand on the difficulties in implementing crosscutting concerns, the solution that contextual composition provides, and problems with this solution.

1.1.1 The Problem with Crosscutting Concerns

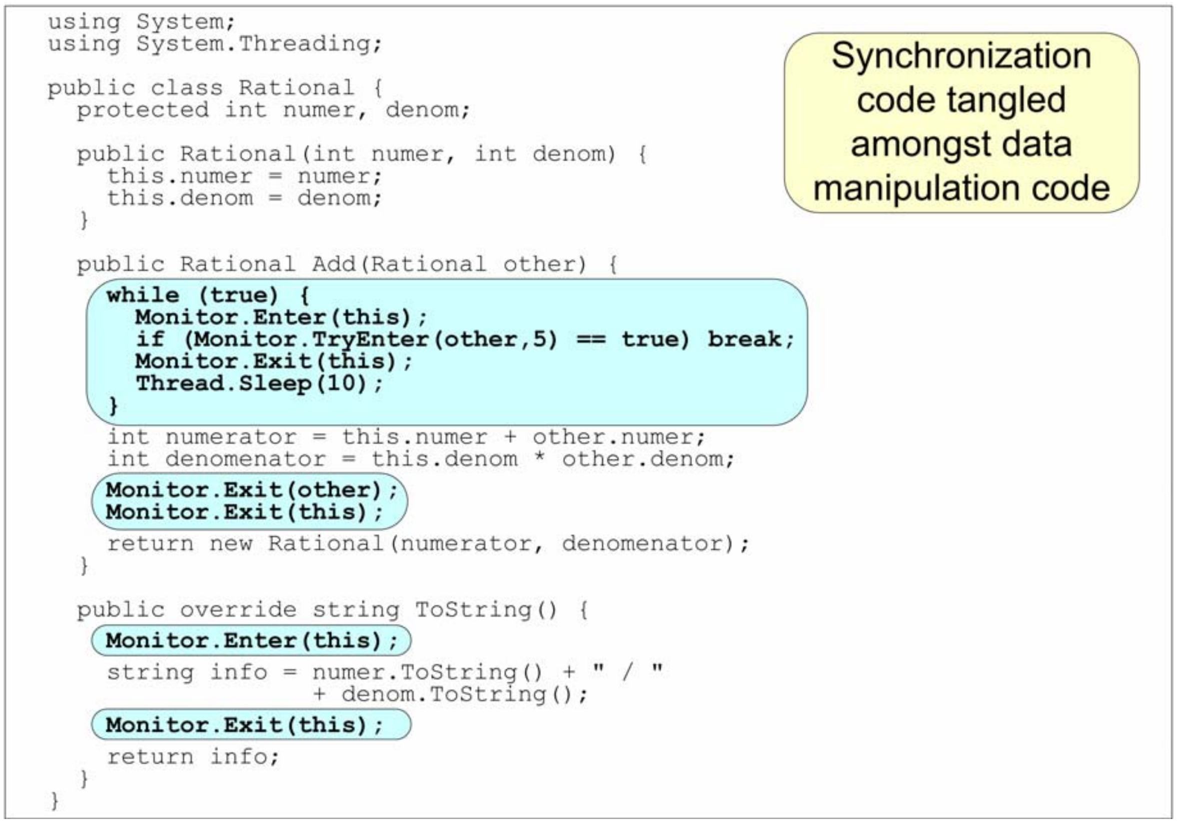
The problem with crosscutting concerns is that they conflict with the distributed nature of software component development and use. Software components emphasise deployment and composition characteristics that allow components provided by one organisation to be combined with components of another by a third-party unrelated to either organisation [Szy'02]. Unfortunately, crosscutting concerns introduce *tangling* [Lop'97] in which the proper binding of two different functionalities requires that their implementations become interlaced. As we will see in this section, tangling integrates a crosscutting concern with the implementation of affected components, and this introduces problems for third-party composition and independent deployment of components.

By definition, software components emphasise an architecture that supports independent deployment and third-party composition of software components. We draw our definition of software components from the area of component-oriented programming, which states that

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [Szy'02]

Independent deployment implies that components will be distributed as separate self-contained entities. This requires that their interactions with other software components as well as their environment must be narrowly specified in terms of what the software component requires for proper functionality as well as what functionality that software component provides. Third-party composition implies a lack of complete knowledge about the components involved in composition. Specifically, "a third party is one that cannot be expected to have access to the construction details of all the components involved." [Szy'02]

Programming paradigms usually leave the encapsulation of crosscutting concerns unaddressed [Tar'99], and so the implementation of these concerns becomes tangled with other functionality in an application. Crosscutting concerns correspond to functionality whose implementation crosscuts the units of encapsulation within an application.



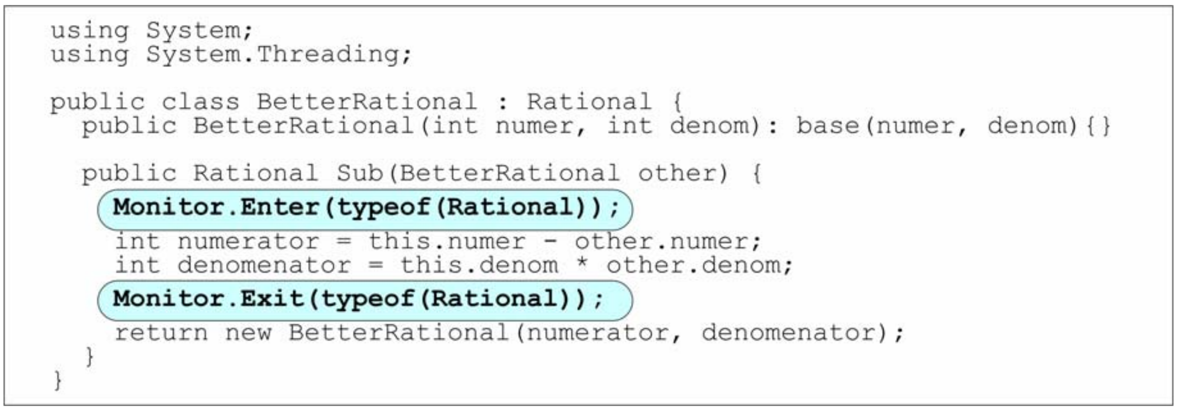
```
using System;
using System.Threading;

public class Rational {
  protected int numer, denom;

  public Rational(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
  }

  public Rational Add(Rational other) {
    while (true) {
      Monitor.Enter(this);
      if (Monitor.TryEnter(other,5) == true) break;
      Monitor.Exit(this);
      Thread.Sleep(10);
    }
    int numerator = this.numer + other.numer;
    int denomenator = this.denom * other.denom;
    Monitor.Exit(other);
    Monitor.Exit(this);
    return new Rational(numerator, denomenator);
  }

  public override string ToString() {
    Monitor.Enter(this);
    string info = numer.ToString() + " / "
                + denom.ToString();
    Monitor.Exit(this);
    return info;
  }
}
```

Figure 1.1: Example of tangling of synchronization functionality for data consistency in a C# class modeling rational numbers.

In object-oriented programming, for instance, crosscutting concerns are "properties or areas of interest" [Elr'01] that normally defy object-oriented modelling [Boo'94], because their implementation does not align to class boundaries. Thus, conceptually simple crosscutting concerns, such as tracing during debugging or synchronization for data consistency, lead to tangling in which code statements addressing a crosscutting concern become interlaced with those addressing other concerns within the application. Indeed, the implementation of synchronization for data consistency provides a particularly good example of tangling due to the difficulty in encapsulating the implementation of data synchronization requirements [Mat'93, Ber'99]. A simple instance of this problem is shown in Figure 1.1, which contains a `class Rational` that models rational numbers. The class is written in C#, and it contains methods to calculate the addition of rational numbers and to convert a rational number to a string. Within these methods, code operating on object data is interleaved with code for data consistency, which is highlighted in the figure. Rational number manipulation is well encapsulated as the addition and string generation methods are limited to manipulating data within `Rational` object boundaries. In contrast, data consistency requirements are decided on an application-wide basis. The data consistency requirements may have to be implemented for other classes in the application besides `class Rational` in the same way, and so the implementation of data consistency is said to crosscut the classes of an application.

With respect to third-party composition, the difficulty with tangling is that it introduces the need to modify the implementation of a component being composed. Take the example of `class Rational` in Figure 1.1, where synchronization characteristics were inserted into the bodies of methods `Add` and `ToString`. Such changes would require an application developer to have access to construction details in order to modify the component's implementation. However, such a requirement violates the notion of third-party composition in which the composer need not have access to component construction details to perform composition.

With respect to independent deployment, the difficulty with tangling is that it forces components to be deployed as sets rather than independently. At issue is the inability to distribute a tangled crosscutting concern independent of the components it affects. Being partly a module [Szy'02], components should be separate units of compilation [Car'97, Fin'98], but tangled functionality cannot be separately compiled. Thus, it is not possible to separate tangled functionalities along software component boundaries. So, the implementation of a crosscutting concern is not contained in a single component, but rather in each of the components with which it is tangled. Take the example of distributing data synchronization. Using object-oriented composition mechanisms, data synchronization must be tangled with the components it affects [Mat'93, Ber'99]. The deployment of this crosscutting functionality then requires that the components containing its implementation be distributed together, because when the implementation of a crosscutting concern is broken up along component boundaries it is possible to introduce errors. For instance, errors in data synchronization are introduced when subtypes of `class Rational` use their own implementation for synchronization. Take the subtype `BetterRational` in Figure 1.2. Although both `class Rational` and `class BetterRational` use monitors, the supertype locks on object instances, while the subtype locks on a separate class-wide object. Thus, the critical sections of `Sub` and `Add` are not mutually exclusive for instances of type `BetterRational`. This would not be the case were the implementation of synchronization consistent in the two types. Guaranteeing such consistency is a matter of using the same synchronization implementation, but doing so no longer allows `BetterRational` to be deployed independently of this implementation of `Rational`.



```
using System;
using System.Threading;

public class BetterRational : Rational {
  public BetterRational(int numer, int denom): base(numer, denom){}

  public Rational Sub(BetterRational other) {
    Monitor.Enter(typeof(Rational));
    int numerator = this.numer - other.numer;
    int denomenator = this.denom * other.denom;
    Monitor.Exit(typeof(Rational));
    return new BetterRational(numerator, denomenator);
  }
}
```

Figure 1.2: Subtype of `class Rational` with incorrect synchronization. Avoiding such errors prevents independent deployment of software components.

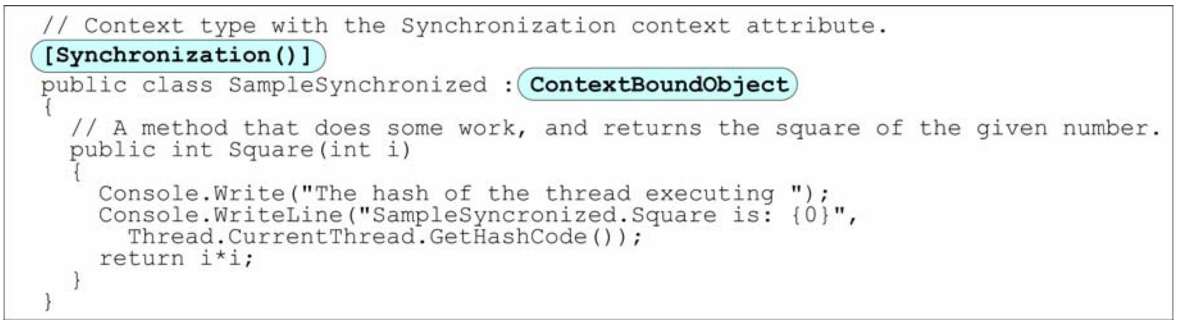
1.1.2 Decoupling with Context Properties

The use of context properties avoids deployment and third-party composition constraints that tangled crosscutting concerns place on components. Context properties are crosscutting concerns implemented by a component framework and composed with component instances with a method interception mechanism. In particular, contextual composition involves binding crosscutting functionality to component instances by declarative selection of context properties, rather than using direct connections, such as method invocation, or derivation mechanisms, such as inheritance [Szy'02]. For example, in container models [Pic'03, Coh'04] context properties correspond to container services bound by a deployment descriptor. Previously, binding crosscutting concerns involved tangling crosscutting functionality with component implementation; however, the contextual composition avoids the need for tangling, which simplifies third-party composition. In these cases, the crosscutting concerns would be distributed with the affected components. However, having the component framework implement crosscutting concerns allows the crosscutting concerns to be addressed consistently across an application without the need for the application’s constituent components to come form the same organisation. This simplifies independent deployment.

Contextual composition uses message interception to compose crosscutting functionality with component instances. In the case of EJB, “contextual composition works by placing a hull around instances and intercepting communication from and to that instance” [Szy'02], and a similar approach is taken in other implementations of contextual composition such as CLR contexts. The interception mechanism is then used to bind object instances to services and resources such as synchronization and transactional properties. For example, the synchronization characteristics required for objects of `class Rational` of Figure 1.1

and `class BetterRational` of Figure 1.2 can be addressed by a context property. Upon interception of an `Add`, `Sub`, or `ToString` invocation, the context property should create a critical section by obtaining locks on object instance and method parameters before forwarding the invocation to the destination method. After the method completes, the context property should release the locks.

The declarative means by which context properties are bound to component instances varies according to the component framework implementing contextual composition. The mechanism for declarative selection can range from deployment descriptors, used with EJB containers [DeM'03], to attribute-based property selection, used with CLR contexts [Mic'04b]. Deployment descriptors identify context property requirements of component instances separate from the component implementation. For example, the context properties of an EJB are described in XML with tags that cross-reference types and type members implemented by a component with the context properties required. In contrast, CLR contexts use attributes to make the association between type implementation and context properties. *Attributes* [Ecm'03b], also called annotations [Blo'03], are a programming-language mechanism for associating additional information with the metadata descriptions of types and their members. Types or type elements become associated with a context property when their implementation is annotated with an attribute corresponding to the context property. This approach is sometimes referred to as attributed programming [Shu'02]. Whether context property selection is by deployment descriptor or by attribute, composing the component implementation to that of the context property is deferred until or after deployment. For example, EJB components are composed with context properties at deployment time typically through code generation [Szy'02]. With CLI, context properties are composed at runtime when objects are instantiated [Szy'02]. By deferring composition with the context property at least until deployment, a component can avoid explicitly addressing the corresponding crosscutting concerns in its implementation [Pic'03]. The `Synchronization` attribute highlighted in Figure 1.3 provides an example of a context property available in the Microsoft .NET implementation of the CLI. This attribute allows a class to define the synchronization characteristics of its objects. The use of this attribute in Figure 1.3 limits the execution of `class SampleSynchronized` methods to one thread at any given time. The figure also highlights the need to inherit from `ContextBoundObject` in order to use the context property, which we discuss in the next section.



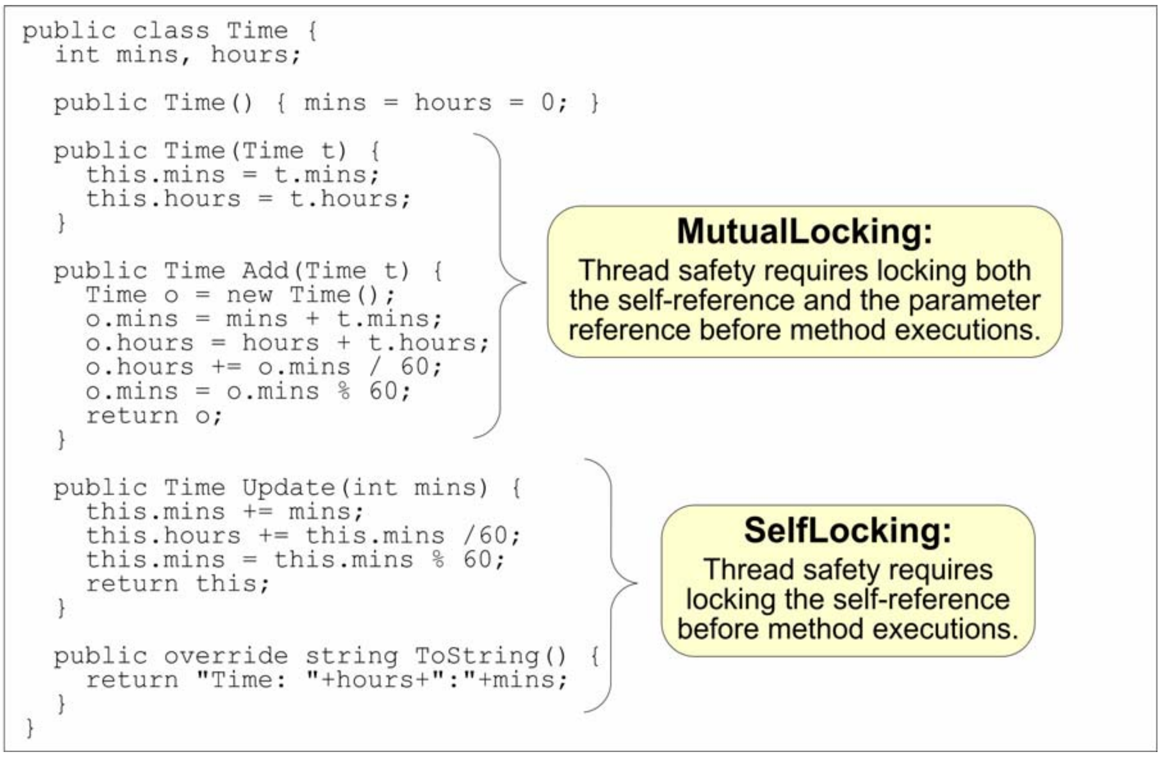
```
// Context type with the Synchronization context attribute.
[Synchronization()]
public class SampleSynchronized : ContextBoundObject
{
  // A method that does some work, and returns the square of the given number.
  public int Square(int i)
  {
    Console.Write("The hash of the thread executing ");
    Console.WriteLine("SampleSyncronized.Square is: {0}",
      Thread.CurrentThread.GetHashCode());
    return i*i;
  }
}
```

Figure 1.3: Use of `Synchronization` attribute available in CLR contexts [Mic'04b].

1.1.3 Limits to Contextual Composition

Unfortunately, the use of contextual composition to address crosscutting concerns is limited by the lack of tailorability problem [Pic'03] as well as the preplanning problem [Tar'99, Cli'00]. Lack of tailorability is an issue, because of the great difficulty in creating context properties to suit the crosscutting functionality required by a component. Preplanning is an issue, because even the most recent contextual composition frameworks, such as EJB and CLR contexts, constrain component architecture to suit the contextual composition mechanism [Shu'02, Coh'04].

Although the preplanning problem originates with design patterns, the same difficulties are faced by components wishing to exploit contextual composition. The *preplanning problem* was observed in the context of design patterns [Gam'94], where it was noted that although the use of patterns allowed for adaptability, they could only be applied if the need for such adaptation was identified during design. We can extend the preplanning problem to contextual composition by observing that the use of context properties by a software component influences the class and object structure implemented by the component, which is also known as the component's architecture [Boo'94]. The underlying difficulty is that contextual composition does not provide a sufficiently general mechanism for binding crosscutting functionality to component instances. For instance, with CLI software components, the class exploiting CLR contexts faces strict inheritance requirements, and so `class SampleSynchronized` of Figure 1.3 was forced to inherit from `class ContextBoundObject`. Likewise, EJB components must implement an interface appropriate to the bean type [Rom'02] along with extensions to the `EJBHome` and `EJBObject` interfaces to allow component instance functionality to be accessed from outside the container. Also, the binding mechanism used in contextual composition constrains component architecture by forcing behaviour being crosscut to be directly activated by messages passing between objects. Thus, intra-object activities such as data access and recursive calls cannot be influenced. In addition to constraining component architecture, these requirements prevent context properties from being used to address crosscutting concerns that are only spotted when the components of an application are assembled. For instance, performance concerns sometimes appear when the final application is composed that can only be solved with result caching described as method memoization [Men'97]; however, by this time it is too late to change component architecture to allow composition with a memoization context property.



```
public class Time {
  int mins, hours;

  public Time() { mins = hours = 0; }

  public Time(Time t) {
    this.mins = t.mins;
    this.hours = t.hours;
  }

  public Time Add(Time t) {
    Time o = new Time();
    o.mins = mins + t.mins;
    o.hours = hours + t.hours;
    o.hours += o.mins / 60;
    o.mins = o.mins % 60;
    return o;
  }

  public Time Update(int mins) {
    this.mins += mins;
    this.hours += this.mins /60;
    this.mins = this.mins % 60;
    return this;
  }

  public override string ToString() {
    return "Time: "+hours+":"+mins;
  }
}
```

MutualLocking: Thread safety requires locking both the self-reference and the parameter reference before method executions.

SelfLocking: Thread safety requires locking the self-reference before method executions.

Figure 1.4: C# source for class `Time` component annotated with its thread safety requirements.

The *lack of tailorability problem* [Pic'03] refers to the inability to extend the set of context properties available to suit the crosscutting functionality required by a component. Specifically, the set of context properties available with MTS, COM+, EJB, and CCM component frameworks is fixed [Szy'02, Pic'03]. Extensibility is available to a limited extent with CLR contexts [Shu'02, Szy'02], but in practice this programming model is ad hoc rather than structured. Unlike other technologies for manipulating crosscutting concerns, such as aspect-oriented programming (AOP) [Elr'01], contextual composition mandates no structured operations for generating or manipulating a crosscutting view of an application. Take for example the synchronization requirements for `class Time` in Figure 1.4. The class requires two synchronization policies depending on whether one or two objects of type `Time` are being manipulated. Since the class is written in a CLI-based language C# [Ecm'03a] and the CLI supports extensible context properties, it is tempting to implement these policies with a context property. While the framework provides an implementation of a monitor to define critical regions, there are is no obvious mechanism for obtaining an object reference from within a context property's implementation for the monitors to lock.

1.2 Solution Domain

Our interest is in overcoming the limits of contextual composition, while retaining its benefits. Such work has been undertaken in the area of aspect-oriented programming (AOP), which recognises context properties as a form of aspect [Kim'02]. In particular, AspectJ2EE [Coh'04] has addressed the lack of tailorability problem with an AOP-based programming model for creating new container services for EJB. An *aspect* [Kic'97] provides a unit of encapsulation that couples the behaviour of a crosscutting concern with a join point specification detail where in component code the behaviour is to be applied. In the context of AOP, *components* [Kic'97] correspond to units of well-encapsulated behaviour, be they source code or binaries. The aspects and components of an application are composed, or *woven*, to produce a single program. This composition is specified in terms of aspect-oriented composition mechanisms [Elr'01]. The use of AOP for contextual composition is contingent on the ability to defer composition of components and aspects at least until deployment time [Pic'03]. This avoids problems with independent deployment that we observed in section 1.1.3 when the implementation of a crosscutting concern is composed with components before they are deployed.

While AOP has been shown to be capable of tackling the lack of tailorability problem, AOP-based alternatives to contextual composition do not eliminate the preplanning requirements imposed, and the use of AOP introduces language dependencies and reusability problems. Recall that AspectJ2EE [Coh'04] avoids lack of tailorability in the context of the EJB container model [Coh'04]. Despite the advances available with AspectJ2EE, only Java classes meeting EJB architectural constraints can access context properties. Another difficulty is that AOP has traditionally been a programming language mechanism [Elr'01], and so its use implies the adoption of a particular language. In contrast, the language-independent characteristics of components [Szy'02] suggest that component interoperability should not be language dependent. As we will see in the next chapter, aspects also tend to be difficult to reuse, as reuse implies customizing the aspect's implementation. The need to rewrite an aspect is problematic, as an alternative to context properties should not add new complexity to the selection of crosscutting functionality.

1.3 Thesis Contribution

This thesis proposes a solution to the problems identified with contextual composition and AOP in the form of aspect-based properties. Aspect-based properties are an alternative to context properties in which crosscutting functionality is modelled with aspects implemented with a pointcut-advice mechanism. The key characteristics of *aspect-based properties* are:

- A pointcut-advice mechanism for defining aspects
- Load-time weaving of aspects with components
- Language-independence
- Support for attribute-based property selection

Each of these characteristics plays a role in allowing aspect-based properties to replace context properties. The pointcut-advice (PA) mechanism allows new crosscutting functionality to be written that can be bound to a richer set of execution points than contextual composition offers, which allows aspect-based properties to address the tailorability problem. While choosing the pointcut-advice mechanism is consistent with its success in implementing context properties in the past [Pic'03, Coh'04], the decision to use this mechanism is based on an analysis of aspect-oriented mechanisms presented in Chapter 2. This analysis noted the PA mechanism to be easier to conceptualise, to offer finer grained crosscutting and to offer better performance. The use of a load-time weaver to compose aspects with components removes the burden of providing support for composition from the software component. Thus, the component inheritance and interface constraints imposed by contextual composition that cause the preplanning problem are avoided. Furthermore, weaving at load-time allows composition of aspect-based properties and components to be deferred until after deployment for consistency with component-oriented programming requirements. Language-independence involves allowing aspects and components to be written in a variety of languages and freely intermixed, regardless of how aspect-based properties are selected. Such characteristics allow aspect-based properties to retain the language-independent nature of component-oriented programming. Attribute-based property selection provides a declarative means of selecting aspect-based properties. Attribute-based property selection provides a simple means of aspect-based property reuse that avoids the need to revise the crosscutting specification of the aspect.

This thesis makes secondary contributions specifically to AOP in the areas of language-independence and the use of attributes. Although AOP technology has been applied to the composition of software components [Coh'04], these technologies have not been demonstrably language-independent in the sense that components and aspects can be implemented in a variety of languages and freely intermixed [Laf'03]. With aspect-based properties we see the first example of an AOP technology that demonstrates language-independence. Also, this thesis addresses reusability with attribute-based property selection. A reuse strategy should decouple aspect binding from aspect implementation so that components can be associated with aspects at deployment time [Pic'03]. Current strategies for reuse of aspects involve providing revised crosscutting specifications. For instance, pointcut-advice mechanisms propose abstract pointcuts [Pic'03, Ras'03, Coh'04] even in systems that allow crosscutting in terms of attributes [Bon'04a]. This thesis addresses reuse with attributes by identifying how they can be used in a fashion consistent with the noninvasive nature of AOP. Specifically, we argue that the use of attributes adheres to the noninvasive emphasis of AOP, as attributes do not actually modify the implementation of component types.

1.4 Orthogonal Issues

This thesis does not address the implementation of existing context properties with aspect-based properties. Work in [Coh'04] has demonstrated the ability for aspects with pointcut-advice semantics to be used to implement EJB container properties, but in our presentation of aspect-based properties we have not attempted to implement a series of existing context properties for aspect-based properties.

This thesis does not touch on aspect frameworks. Work in [Ras'03] describes a framework for persistence consisting of several aspects. The need to use multiple aspects to properly model crosscutting concerns such as persistence is not dealt with in our work on aspect-based properties.

Finally, we leave the issue of mediating the composition of different crosscutting concerns unaddressed. Organising the application of multiple aspects, such that each does not adversely affect the others, is an issue in contextual composition, referred to as the composability problem [Szy'02], as well as aspects [Kni'01a] to which aspect-based properties make no contribution.

1.5 Evaluation Criteria

The evaluation should determine if aspect-based properties are realistic to adopt for component-oriented programming. Adoption is an issue of whether programmers can realistically migrate from context properties to a programming model involving aspect-based properties. Adoption requires that users not have to abandon existing programming techniques [Coh'04]. We understand this to mean that developers should not have to abandon the programming languages that they current use to implement components to take advantage of aspect-based properties, nor should the developer have to abandon existing components. This is a matter of ensuring that aspect-based properties properly support language-independence, and that crosscutting specifications can, if necessary, be tailored to existing components. Since AOP is a reasonably novel approach [Elr'01], we can expect that existing programmers do not have a strong knowledge of AOP mechanisms, and so a second facet of adoption is whether aspect-based properties can be used without the need to modify an aspect. This is a matter of demonstrating the attribute-based property selection of aspect-based properties is available in a manner consistent with language-independence.

The second evaluation problem is to determine if aspect-based properties make tangible improvements on contextual composition. The tangible benefits of aspect-based properties concern their ability to solve the preplanning and tailorability problems. With respect to preplanning, we want to see that aspect-based properties place less design constraints on software components than context properties. This comparison should involve a simple example, in which custom crosscutting functionality is written for a software component both as a context property and as an aspect-based property. This comparison requires a component framework that provides a mechanism for writing new context properties such as CLR contexts do for the CLI. With respect to tailorability, a minimal test is to illustrate that custom aspect-based properties can be written. A more interesting test is to illustrate that custom aspect-based properties can address problems beyond the scope of context properties. Since AOP originated in part to lower application execution times [Kic'97], we would like to see a custom aspect-based properties used in order to lower the execution time of an application. Also, we would like to see aspect-based properties not introduce any more execution overhead than context properties, and ideally for them to introduce less.

1.6 Thesis Overview

The remainder of this thesis provides a grounding in aspect-oriented technology and characterises aspect-based properties. Chapter 2 reports on the state of the art in AOP starting with an introduction to important AOP terminology, which is followed with the taxonomy of the principle aspect-oriented mechanisms. Finally, the implementation issues with software component weaving are summarised. Chapter 3 presents aspect-based properties in the context of their programming model. The programming model is first presented in terms of the developer roles involved with using aspect-based properties for application development and the products of these roles. Next, the chapter focuses on the aspect model used to implement crosscutting functionality with aspect-based properties in which crosscutting specifications are written in XML and behaviour implemented with a component type. Chapter 4 provides implementation details for a prototype weaver, called Weave.NET, that supports aspect-based properties for the CLI component platform. This chapter provides details on why the CLI platform was targeted, how the XML schema for crosscutting specifications was devised, how the weaving APIs are implemented, and how they are integrated with the execution environment to provide load-time weaving. Chapter 5 evaluates aspect-based components based on the criteria described in section 1.5. Chapter 6 summarises the thesis and its contributions, and then it points out future avenues of research.

1.7 Summary

In this chapter we introduced contextual composition as a declarative means of binding crosscutting functionality with components. Contextual composition suffers from the lack of tailorability problem in that the set of context properties is difficult if not impossible to extend. The preplanning problem is another issue, and it refers to the inability to use context properties without influencing component architecture. These issues can be solved with aspect-oriented programming (AOP). However, aspect-based properties are required to overcome language dependency and reusability issues with AOP technology. Aspect-based properties are characterised by language-independent pointcut-advice semantics, support for attribute-based property selection and a load-time weaving architecture. Evaluation criteria will concern the adoptability of aspect based properties and their ability to solve the lack of tailorability and preplanning problems.

Chapter 2 State of the Art

"Originality is the fine art of remembering what you hear, but forgetting where you heard it"
–Laurence Peter

This chapter surveys the aspect technology used to address crosscutting concerns in software. In our analysis, we find that among the canonical aspect-oriented mechanisms, it is the pointcut-advice mechanism that best addresses the tailorability and preplanning problems with contextual composition. A pointcut-advice mechanism allows the aspect to directly influence execution with fine-grained aspects, and its use does not impose restrictions on the programming model for components. Unfortunately, the reuse strategies for aspect-oriented mechanisms rely on being able to customise the aspect, whereas reuse of a context property leaves the property unchanged. Also, interoperability is at the language level, leaving aspect users to define their component to suit the aspect-oriented language. As for retaining the independent deployment characteristics of components, we find load-time weaving can address the need to keep crosscutting functionality separate until after deployment without the need to consider the implementation language of components or aspects.

To provide an in-depth understanding of AOP, we describe the canonical AOP mechanisms in terms of their aspect model rather than simply looking at the operations these mechanisms make available for expressing aspects. An aspect model describes the crosscutting semantics of an aspect technology in terms of its join point model, its means of identifying join points and its means of modifying application semantics at these join points [Kic'01a]. A join point model specifies the elements of an application to which an aspect can bind to or can influence. Join points are manipulated as sets rather than individually, and to create these sets there must be some means of identifying join points. Finally, manipulating the sets of join points requires a means of modifying application semantics on a set-wide basis. In contrast, we view aspect-oriented mechanisms as the abstractions used to express an aspect. Whereas mechanisms define operations with which an aspect is implemented, an aspect model provides details on the semantics of aspects and an insight into how the operations provided by the mechanism are to be used.

We have identified five canonical aspect-oriented mechanisms, which are as follows:

- The pointcut-advice (PA) mechanism exemplified by the work of the AspectJ Team [Asp'00]
- Class composition exemplified by the work of the MDSOC Project [IBM'00a]
- Object-graph traversal exemplified by the work of the Demeter Project [Lie'00]
- Open class composition, which originated with mixins [Moo'86], but is exemplified by the inter-type declaration semantics of AspectJ
- Composition Filters (CF) object model extensions available with tools such as ComposeJ [Car'01]

The first four mechanisms were identified as aspect-oriented in work to develop a definition of what makes a modular crosscutting mechanism aspect-oriented [Mas'03]. The CF Model appears frequently as a prominent aspect-oriented mechanism in AOP community literature, such as [Ber'01]. The CF mechanism is also of interest in that it provides a tailorability mechanism to similar context properties in that it relies on method interception and offers to avoid the preplanning problem with contextual composition.

The rest of this chapter establishes the fundamentals of aspect technology, critiques the five canonical mechanisms, and examines implications of component composition for weave time. In section 2.1, we clarify what is understood to be an aspect and an aspect model. Also, the concept of obliviousness is examined along with it consequences for the reusability of aspects and the complexity of component maintenance. Section 2.2 provides an overview of AOP mechanisms and current approaches to language-independence. Each AOP mechanism is then discussed in-depth. Background details summarise the technology’s origins and goals. The mechanism is characterised in terms of the underlying aspect model, and where possible programming details are provided. The mechanism is analysed for its usefulness in solving the problems of context properties, providing language-independence, and simplifying reuse. In section 2.3, we examine what requirements component composition places on aspect weavers and we characterise a common solution to these issues in the form of load-time weaving.

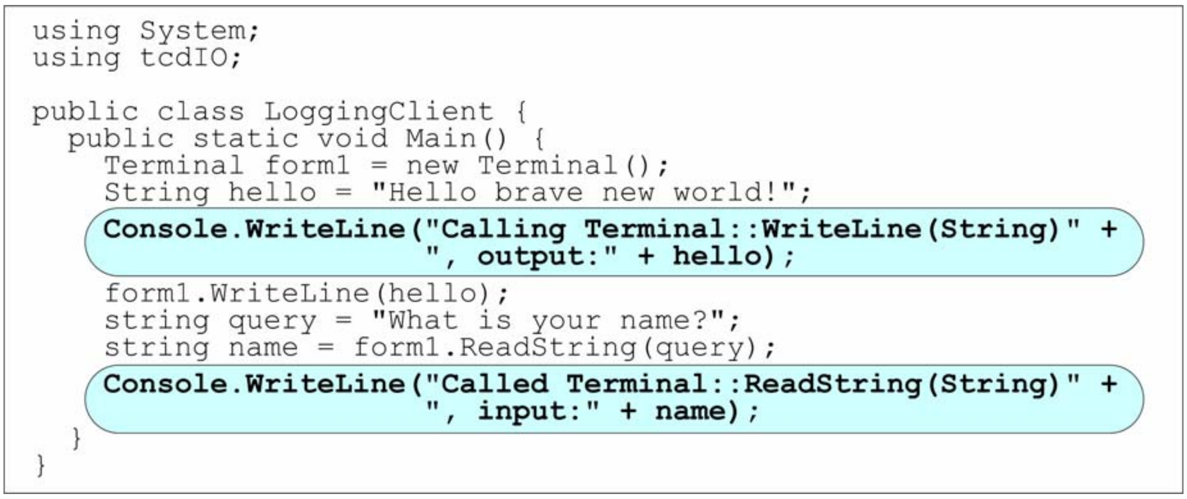
2.1 Principles of Aspects

In section 1.2, the aspect was presented as a unit of encapsulation motivated by the need to avoid software engineering problems surrounding tangling. So far, we have mentioned five mechanisms with aspect-oriented characteristics. What these technologies share is the ability to implement functionality that crosscuts an application in a well-encapsulated manner. What differentiates these technologies is what elements of the application are crosscut and how crosscutting functionalities are implemented. These details are given by the aspect model of a mechanism. The aspect model describes the semantics underlying a mechanism. With the model in mind, it is possible to exploit a particular AOP mechanism for writing aspects. These aspects address tangling based on the component to which they are applied being oblivious to the application of the aspect. Obliviousness separates implementation of the aspect and component by dictating that aspect-related artefacts should not be placed in the component implementation.

In the following subsections, we go into detail on aspects. First, we characterise aspects and point to the common characteristics of aspect-oriented technologies. Next, we examine the concept of an aspect model in greater detail. The constituent elements are discussed one by one with the aid of an example based on the PA aspect-oriented mechanism. Finally, we look at the consequences of obliviousness for the specification of an aspect’s crosscutting semantics.

2.1.1 Aspects

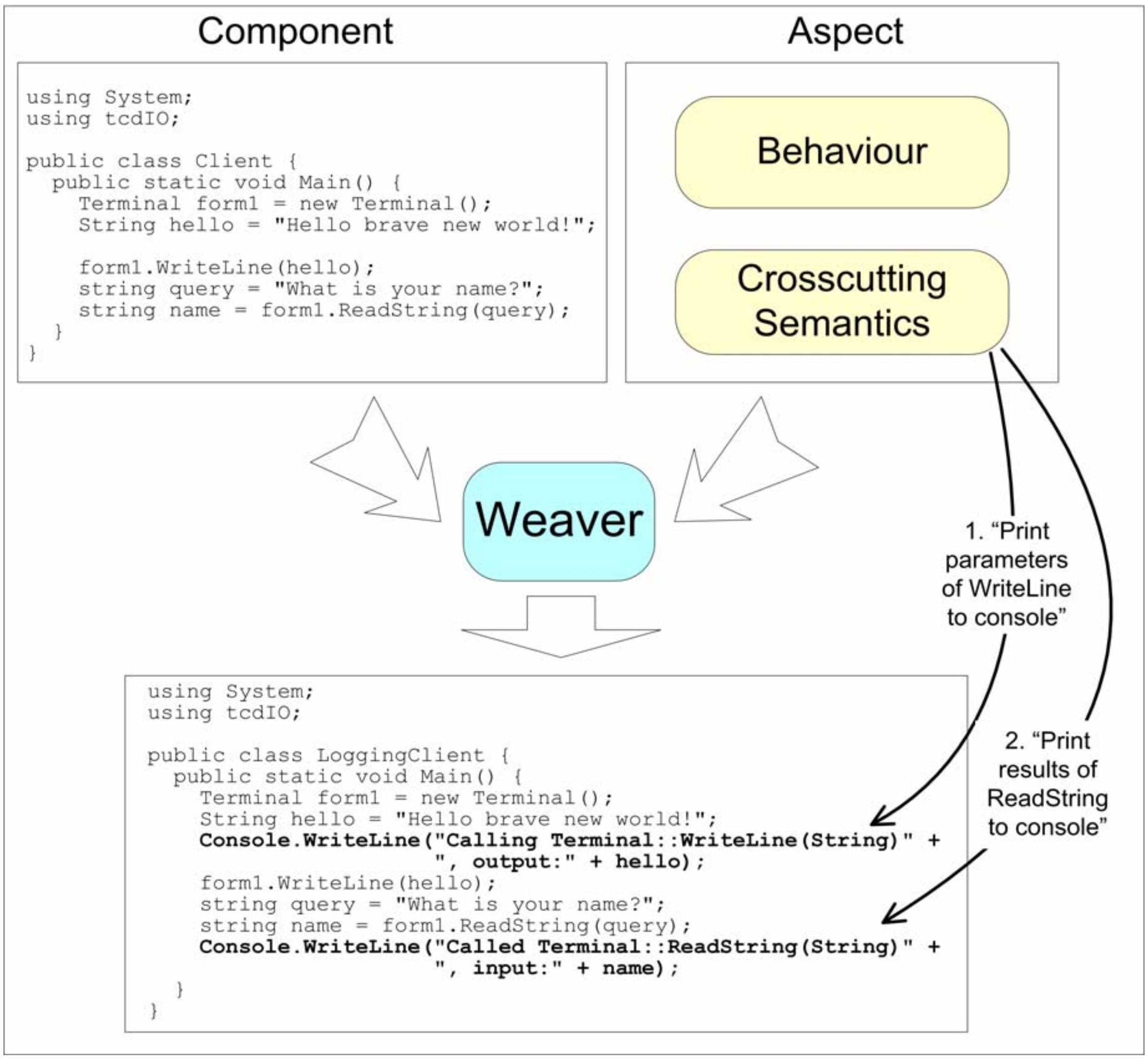
At first glance the notion that aspects allow the encapsulation of crosscutting concerns may seem paradoxical. Recall from section 1.2 that an *aspect* [Kic'97] provides a unit of encapsulation that couples the behaviour of a crosscutting concern with a join point specification that details where in component code the behaviour is to be applied. It would appear contradictory that an aspect can encapsulate crosscutting concerns if crosscutting concerns by definition cannot be encapsulated. This apparent contradiction is referred to in the literature as the *aspectual paradox* [Lie'99], and it is addressed by pointing out that there is a primary decomposition paradigm within a programming language and that it is from the point of view of this primary decomposition paradigm that crosscutting concerns are observed. For example, in OO systems data is encapsulated along object boundaries, and we tend to think of such systems in terms of their objects. However, the data hiding properties of object boundaries would hinder the implementation of functionality such as persistence, which requires direct access to the data of multiple objects and multiple types of objects. In solutions for persistence based on aspects [Soa'02, Ras'03], persistence is still a crosscutting concern, because persistence still crosscuts the data hiding boundaries of objects in the application. Similarly, the aspectual paradox is resolved by considering that the mechanisms proposed by AOP allow the manipulation of arbitrary sets of program elements, such as data members of objects, and these program elements are normally encapsulated separately with other units of decomposition such as object boundaries in object-oriented programming. From this view, AOP can be seen as breaking the "tyranny of the dominant decomposition" [Tar'99] paradigm of a particular programming language.



```
using System;
using tcdIO;

public class LoggingClient {
  public static void Main() {
    Terminal form1 = new Terminal();
    String hello = "Hello brave new world!";
    Console.WriteLine("Calling Terminal::WriteLine(String)" +
                      ", output:" + hello);
    form1.WriteLine(hello);
    string query = "What is your name?";
    string name = form1.ReadString(query);
    Console.WriteLine("Called Terminal::ReadString(String)" +
                      ", input:" + name);
  }
}
```

Figure 2.1: Tangled implementation of logging crosscutting behaviour.

Aspects eliminate tangling by supporting modular crosscutting. For our purposes, modular crosscutting describes the ability of an aspect to specify, in a well encapsulated fashion, its bindings to the final application. This final application is generated when a weaver combines any number of programs together to create a single combined computation [Mas'03]. Take the example of Figure 2.1 in which logging calls are inlined with the implementation of a simple I/O program. In contrast, Figure 2.2 separates logging calls from the I/O program by placing them in an aspect. This requires the aspect include logging behaviour as well as specifications for its integration with the final application. In Figure 2.2, these semantics are interpreted by the weaver, and they result in a program with behaviour identical to that of the tangled original. In this case, crosscutting is modular because its specification is encapsulated with the aspect.


Component
Aspect
Behaviour
Crosscutting Semantics
Weaver
1. "Print parameters of WriteLine to console"
2. "Print results of ReadString to console"

```
using System;
using tcdIO;

public class Client {
  public static void Main() {
    Terminal form1 = new Terminal();
    String hello = "Hello brave new world!";

    form1.WriteLine(hello);
    string query = "What is your name?";
    string name = form1.ReadString(query);
  }
}
```

```
using System;
using tcdIO;

public class LoggingClient {
  public static void Main() {
    Terminal form1 = new Terminal();
    String hello = "Hello brave new world!";
    Console.WriteLine("Calling Terminal::WriteLine(String)" +
                      ", output:" + hello);
    form1.WriteLine(hello);
    string query = "What is your name?";
    string name = form1.ReadString(query);
    Console.WriteLine("Called Terminal::ReadString(String)" +
                      ", input:" + name);
  }
}
```

Figure 2.2: Visualization of weaver

2.1.2 Aspect Model

The concept of an aspect model introduces a structured means of describing a particular aspect-oriented mechanism. The defining elements of an aspect model correspond to the characteristics that allow an aspect-oriented mechanism to crosscut an application. For these elements, we turn to an investigation into the commonality among recognised aspect-oriented mechanisms discussed in [Mas'03]. This investigation "produces a clear three part characterization of what is required to support crosscutting structure: a common frame of reference that two (or more) programs can use to connect with each other and each provide their semantic contribution." [Mas'03] This statement can be confusing to readers, as the three parts that characterise the crosscutting nature of an aspect-oriented mechanism are not presented in enumerated form. The first part of the characterization is the "common frame of reference", which allows aspects to refer to elements of the final application, rather than other programs that are composed to create the final application. In terms of an aspect model, this common frame of reference corresponds to a *join point model*, which identifies the possible connection points that an aspect can potentially exploit to influence the final application. Thus, the join point model corresponds to the "common frame of reference" referred to by an aspect in its crosscutting specification. The second element of the characterisation is a description of how the aspect-oriented mechanism refers to program elements. In terms of an aspect model, this second element is the *means of identifying join points*, which gives an aspect mechanism its ability to "connect" to the final application. The third element is a description of what can be done by an aspect to program elements in the final application in order to make a "semantic contribution". In terms of an aspect model, this third element is the *means of modifying join point semantics* that an aspect-oriented mechanism provides.

The following subsections elaborate on the function of each element of an aspect model. To make our description concrete, we map the functionality of a pointcut-advice (PA) mechanism to aspect model elements. This analysis of a PA mechanism is cursory, and a more in-depth explanation is presented in section 2.2 when the canonical PA mechanism implemented by AspectJ is analysed.

2.1.2.1 Join Point Model

A crosscut can be thought of as an arbitrary collection of potentially unrelated program elements. By unrelated, we mean that there is no existing grouping by which we can manipulate these program elements from a single point. Take the example of a set of types in an object-oriented language supporting inheritance. If the types in the set are all subtypes of a common class, adding fields or methods to all types in the set is a matter of making changes to the common super-type. Using this single point of change is visualised on the left-hand side of Figure 2.3, where modifications to all types correspond to a change to one class. For types not related by inheritance, the modification involves multiple points of change, which crosscut several class structures. This scenario is visualised on the right-hand side of Figure 2.3. Our example of type modification involves the static structure of a program, but the execution flow of the program can be crosscut as well. Consider applying logging to method invocations. We presented this example in Figure 2.1 and again in Figure 2.2. If logging is applied to invocations of a single method, it is possible to revise the method containing the invocations to include logging semantics. If logging is to be applied to invocations in multiple methods contained in multiple types, the addition of logging will involve changes to methods throughout a program. In the example of adding type members as well as logging invocations, the crosscut is the set of program elements that are being manipulated, while the join points are the constituent elements of this set. While type structure and method behaviour were being modified in the examples

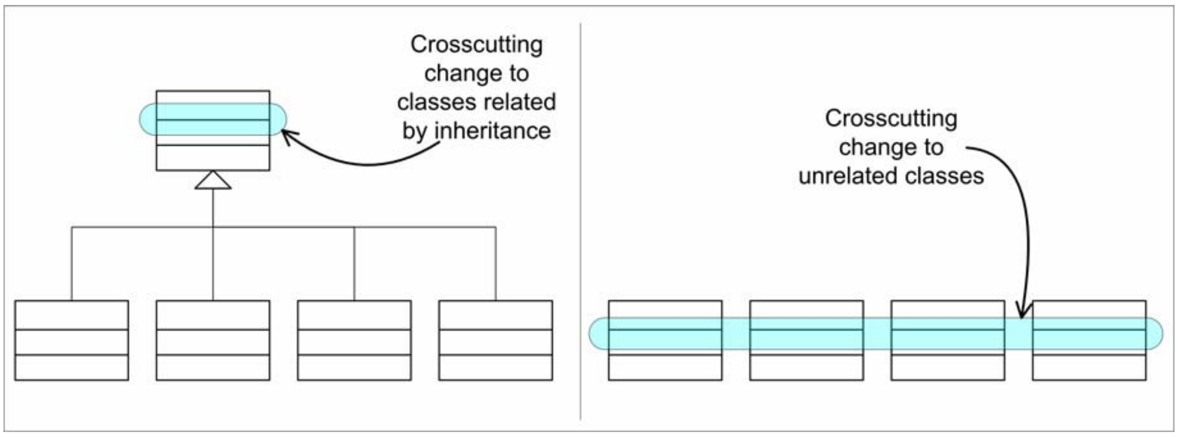

Crosscutting change to classes related by inheritance
Crosscutting change to unrelated classes

Figure 2.3: Visualization of crosscutting types.

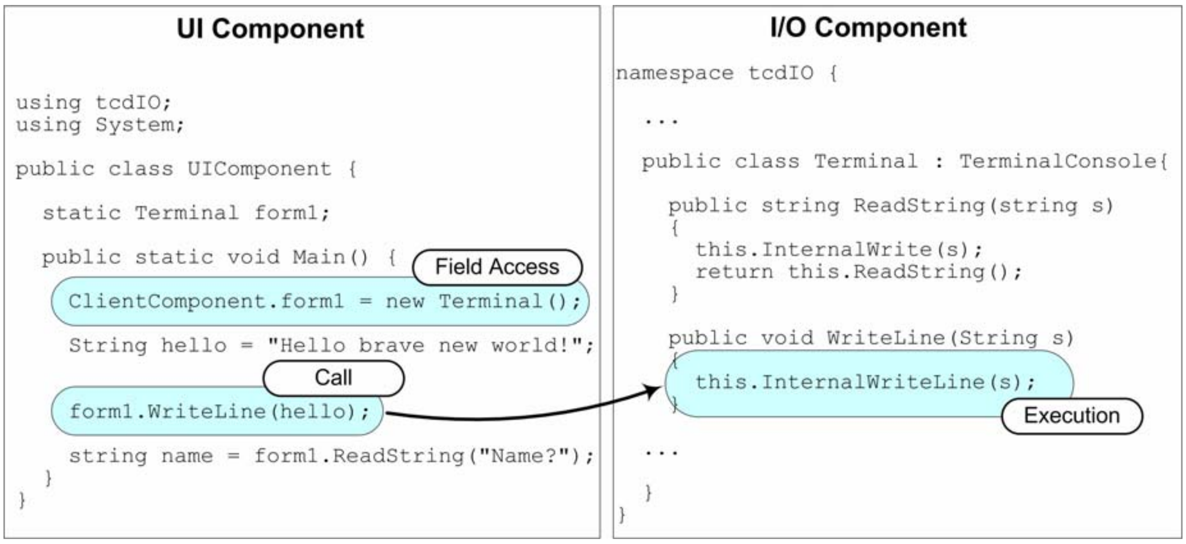
described, the program elements that are made available for manipulation vary according to the AOP mechanism being used.

Even though aspects and the components to which they are applied were introduced as separate entities, aspects have the ability to crosscut join points that they implement and join points implemented by other aspects as well as those implemented by components. An important consequence of the three part characterisation of an aspect-oriented mechanism is that the set of join points manipulated by an aspect comes from the woven application, and not from a limited set of the programs being woven. Thus, the join points available for manipulation by an aspect can include join points implemented by the aspect itself as well as join points implemented by other aspects. In practice, self-referential crosscutting semantics may be limited in order to simplify weaver design.

Table 2.1: Categorization of dynamic join points.

Join point category
Execution
Call
Field access

In the case of a PA mechanism, join points correspond to “well-defined points in the execution flow of the program” [Kic'01a]. These join points are referred to as *dynamic join points* in PA mechanism terminology [Kic'01a]. Dynamic join points fall into three categories: execution join points, call join points and field access join points [Asp'02, Laf'03]. This categorization is summarised in Table 2.1, but a clearer understanding can be gained from Figure 2.4 where example implementations of each join point kind are presented. Execution join points roughly correspond to the execution of a block of code, as opposed to a call or dispatch to that block. In the simplest case, the block may correspond to the body of a method. However, finer distinctions exist when it comes to the execution of exception handlers and the sequence of constructor executions and data member initializations during object creation. Call join points are present on the calling side of a method invocation or when the `new` operator is used for object construction. The final category of join point is that of field access, which corresponds to a read or write access to a data member of an object or class.



```
using tcdIO;
using System;

public class UIComponent {

  static Terminal form1;

  public static void Main() {
    ClientComponent.form1 = new Terminal();
    String hello = "Hello brave new world!";
    form1.WriteLine(hello);
    string name = form1.ReadString("Name?");
  }
}
```

```
namespace tcdIO {

  ...

  public class Terminal : TerminalConsole{

    public string ReadString(string s)
    {
      this.InternalWrite(s);
      return this.ReadString();
    }

    public void WriteLine(String s)
    {
      this.InternalWriteLine(s);
    }

  ...

  }
}
```

Figure 2.4: Examples of join points for a pointcut-advice aspect-oriented mechanism (source in C#).

2.1.2.2 Means of Identifying Join Points

AOP mechanisms provide a means of identifying join points in order to create sets of join points that crosscut an application. Tangling occurs when modifications to several join points are inlined with the implementation of these join points. An aspect centralises changes to join points by specifying the changes once and applying them to a set of join points. In order for this set to be created, the aspect must have a means of identifying join points. Moreover, the means of selecting join points must allow arbitrary join points to be selected. Arbitrary join point selection was demonstrated in the right-hand side of Figure 2.3 in which classes unrelated by inheritance were selected for the same modifications.

It is useful in the characterising the reusability of an aspect-oriented mechanism to distinguish name-based crosscutting from property-based crosscutting. These terms originate from their use in the context of PA aspect mechanisms [Kic'01a]. In this context, *name-based crosscutting* involves identifying individual join points by complete implementation information for that join point. In the case of a PA model, these include the containing type, identifying name and signature details of the join point. In contrast, *property-based crosscutting* refers to groups of join points by their commonality such as a shared containing type, some naming convention, or common parameter types in the case of methods. We can generalise these categories to the whole of AOP by considering name-based crosscutting to identify join points based on complete details of the join point's implementation and property-based crosscutting to use properties that distinguish groups of related join points.

The precise details used in distinguishing join points will vary according to the join point model of the aspect-oriented mechanism. For a PA mechanism, a name-based crosscutting specification of the join points identified in Figure 2.4 would involve citing the join point type as well as declaration details of the join point's implementation. In the case of the call join point, this would involve identifying the join point of interest as being a call join point that invokes a method in type `Terminal` with the following declaration: `void WriteLine(String s)`. Property-based crosscutting would take into account some other commonality. While the field access and call join points in Figure 2.4 share no naming conventions, they could be grouped according to the type containing their implementation. For a second example, let us turn to the object graph traversal aspect-oriented mechanism offered by the Demeter tool. In contrast to the PA mechanism offered by AspectJ, Demeter supports crosscutting of the object graph within an object-oriented application, and the join points are traversal to and visitation of objects in this object graph. For this aspect-oriented mechanism, a form of name-based crosscutting would be to identify objects according to absolute paths between objects in the object graph, whereas property-based crosscutting would locate objects in terms of relative details such as neighbour node type.

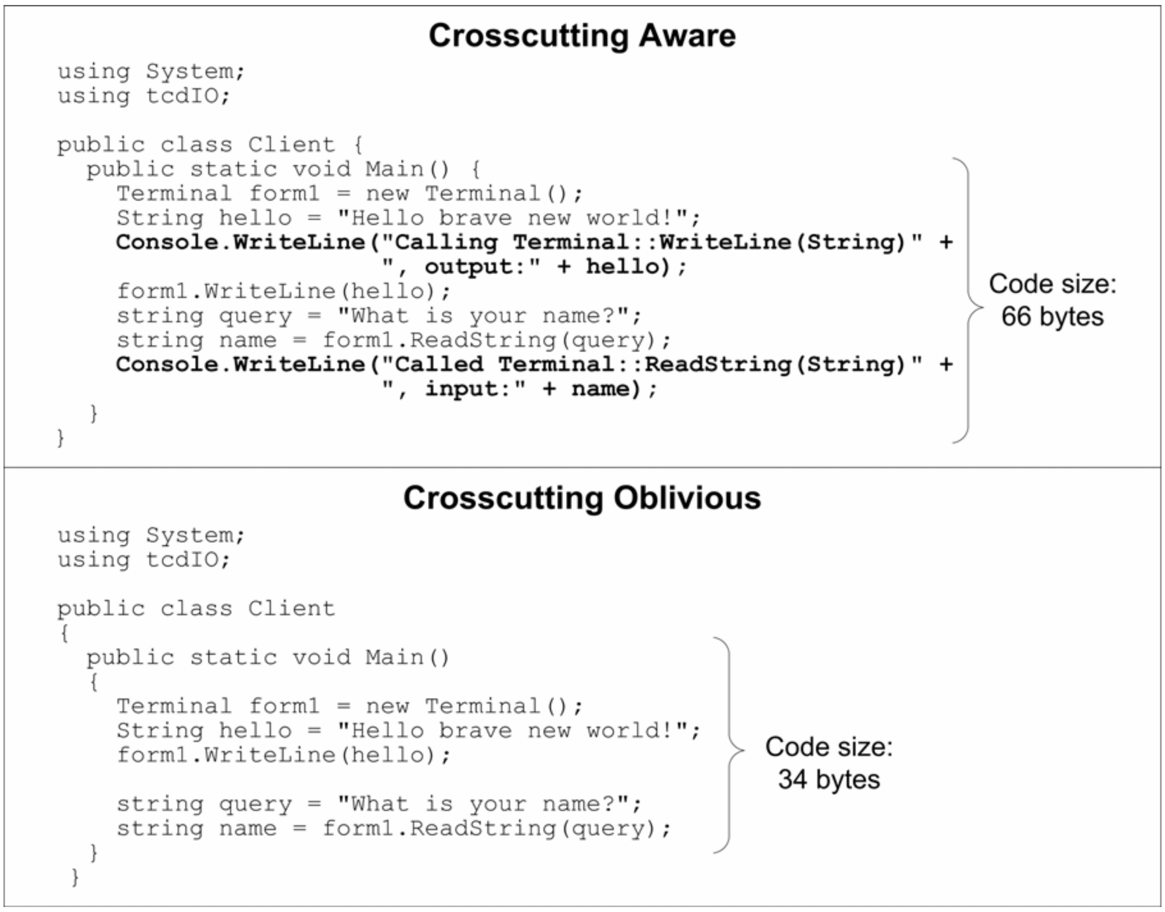
2.1.2.3 Means of Modifying Join Point Semantics

The third element of an aspect model, the means of modifying join point semantics, includes structural as well as behaviour modifications that an aspect exploits to influence the woven application. In earlier work, the third element of an aspect model was described as the means of modifying join point behaviour [Laf'03]. This kind of modification is typical of a PA mechanism in which aspects influence join points in terms of advice. Broadly speaking, an advice statement associates a set of join points with aspect behaviour and specifies how to execute this behaviour relative to the execution of the join points. The behaviour is either executed before, after, or in place of the join point. The later case is referred to as around advice, and around advice typically retains the capability to activate the join point. However, viewing aspects as only influencing behaviour does not allow an aspect model to characterise recognised aspect-oriented mechanisms such as mixins [Moo'86]. Crosscutting occurs when the same mixin's behaviour is applied 'as is' to a number of classes in an application. Normally, a mixin makes additive changes to a type. Such changes do not modify existing execution join points, and so the changes made using mixins cannot be characterised purely in terms of modifying join point behaviour. By stating the third element of a join point model to be a means of modifying join point semantics, we can take into account purely structural changes such as those offered by mixins.

2.1.3 Obliviousness

Obliviousness hides details of crosscutting functionality from components being crosscut to simplify the task of programming these components. *Obliviousness* [Fil'00] is a property of aspect-component composition. Obliviousness is achieved when the component implementation shows no explicit evidence of the application of aspect behaviour. In these circumstances, the developer of the component is oblivious to the application of an aspect. Obliviousness allows for a separation of concerns between aspects and the components to which they are applied. *Separation of concerns* as coined by Dijkstra [Dij'76] refers to compartmentalizing the different issues at play in a problem so that each can be solved separately and independently [Ber'99]. In this case, obliviousness brings about a separation between the crosscutting functionality being provided by an aspect and the implementation details of components to which the aspect is applied. Thus, the component need not account for the crosscutting functionality in its implementation, nor need it provide hooks for binding crosscutting functionality to component behaviour. For example, recall the two versions of component code from Figure 2.1 and Figure 2.2 in which logging properties were being added to a simple I/O program. These two implementations of the I/O program are pictured in Figure 2.5. The top version is taken from Figure 2.1, and it includes logging calls tangled with I/O instructions. The bottom version is taken from Figure 2.2, where logging was added to the component using a weaver and an aspect. With the crosscutting functionality removed, the compiled component's code size drops by about half. From a software engineering perspective, this should be an indication that the component is addressing a simpler problem, which should make it easier to design and implement.

The advantages of obliviousness have led to a focus on noninvasive composition in AOP. At its extreme, obliviousness implies that aspects will leave no artefacts at the join points they affect. When using the term artefact, we mean any sort of manual modification required to allow the component to accommodate an aspect. To meet this requirement, the aspect-oriented mechanism being used must allow noninvasive adaptability. “By non-invasive adaptability, we mean the ability to adapt a component or an aspect without manually modifying it.” [Cza'00] For example, the component described in the bottom of Figure 2.5 provides no indication of where the logging methods are going to appear in the woven version. Note that this concept of noninvasive composition was originally introduced when AOP was viewed as a compile time activity, where source code of both aspect and component was available. In the context of software components, it is the software components and not their source code that is being composed. In this case, invasiveness concerns the artefacts that the aspect leaves in a software component’s source code.



Crosscutting Aware

```
using System;
using tcdIO;

public class Client {
  public static void Main() {
    Terminal form1 = new Terminal();
    String hello = "Hello brave new world!";
    Console.WriteLine("Calling Terminal::WriteLine(String)" +
                      ", output:" + hello);
    form1.WriteLine(hello);
    string query = "What is your name?";
    string name = form1.ReadString(query);
    Console.WriteLine("Called Terminal::ReadString(String)" +
                      ", input:" + name);
  }
}
```

Crosscutting Oblivious

```
using System;
using tcdIO;

public class Client
{
  public static void Main()
  {
    Terminal form1 = new Terminal();
    String hello = "Hello brave new world!";
    form1.WriteLine(hello);

    string query = "What is your name?";
    string name = form1.ReadString(query);
  }
 }
```

Figure 2.5: Components oblivious (bottom) and aware (top) of crosscutting written in C# and annotated with compiled code sizes for the `Main` method.

Unfortunately, noninvasive adaptation of components with aspects makes it harder to write and reuse aspects. According to an evaluation of AOP [Mur'99], AOP is much simpler if aspect code has a well-defined effect on particular points of code, which suggests a development goal is to make the crosscuts as specific as possible. To do so with noninvasive adaptation requires that the aspect's crosscutting specifications be tailored to the component to which the aspect is being applied. In the case that an aspect is being reused, this involves updating the sets of join points that the aspect manipulates to reflect the new component. For example, if the logging aspect of Figure 2.2 were revised to be applied to the code in Figure 2.6, the crosscutting semantics would have to be restated. The `ReadInt` and `ReadBool` calls print questions to the console, so we would want to log their invocation parameters before the calls, and record the values returned by the user by logging the call results. Using name-based crosscutting, each join point that is manipulated by an aspect must be cited specifically. In the case of Figure 2.6, two join points are cited and so name-based crosscutting is manageable. However, when larger applications are crosscut, the number of citations can be difficult to manage. Moreover, the aspect cannot be reapplied to a different application, as it cites specific details of the current one. Property-based crosscutting can substantially reduce the number of terms required to identify the join points affected by an aspect by citing the commonality that join points share rather than citing join points individually. In the case of Figure 2.6, we could apply logging to all calls to methods in the `Terminal` class. However, with property-based crosscutting it is easy to inadvertently select or forget to select join points, because the join points being selected are not apparent from the application source either to the component or aspect programmer. The internals of the `tcdIO` library [Cah'02][1] were alluded to in Figure 2.4, and they are repeated in Figure 2.7 where we see that the class makes internal calls to I/O functions. Property-based crosscutting that selects all calls to a method in `class Terminal` will log these internal calls which are not of interest. Thus, with property-based crosscutting we trade the verboseness of name-based crosscutting for an error prone mechanism. Moreover, property-based crosscutting with obliviousness in mind still has limited reuse. Property-based crosscuts are still coupled to the component implementation, and so reuse requires that component programmer and aspect writer coordinate to guarantee that components implements types with the commonalities identified by the property-based crosscuts. However, the implementation of types with such commonalities in mind violates strict obliviousness.

[1] The I/O tcdIO library was implemented for the CLI platform for teaching purposes as described in [Cah'02] Cahill, V. and Lafferty, D. *Learning to Program the Object-Oriented Way with C#.* Springer-Verlag UK, London, 2002. and is available for download from http://csharp.dsg.cs.tcd.ie

```
using System;
using tcdIO;

public class Client {
  public static void Main() {
    Terminal form1 = new Terminal();
    string query = "What is your age? ";
    int age = form1.ReadInt(query);

    query = "Is " + age + "old?";
    string name = form1.ReadBool(query);
  }
}
```

Figure 2.6: New I/O program requiring revised crosscutting semantics to log console access.

In the case of components being maintained, noninvasive adaptation makes code difficult to reason about as well as to refactor. When reasoning about components, the difficulty is that it is not possible to determine from component source whether or not additional or modifying functionality is being provided by aspects. Thus, the observation that a more explicit mechanism for alerting users to an aspect's presence would ease maintenance [Lip'99]. Referring to Figure 2.6, it is impossible to know that the code is being influenced by a logging aspect from component source alone. Logging does not influence component behaviour significantly, so may be possible to reason about the component in this example without the knowledge of aspects being applied. Where the aspect implements security or synchronization constraints, its influence on component behaviour must be considered. In these cases, noninvasiveness may lead to a misinterpretation of component semantics. In terms of refactoring, noninvasive adaptations tend to break when the implementation of the component is modified [Tou'03]. Essentially, noninvasive crosscutting specifications make assumptions about the structure of the application when join points are identified. For instance, name-based crosscuts mirror join point implementations by including details such as the containing type in the case of method calls or executions, or the destination method name in the case of messages being passed to an object. Refactoring revises the application structure, which breaks these assumptions.

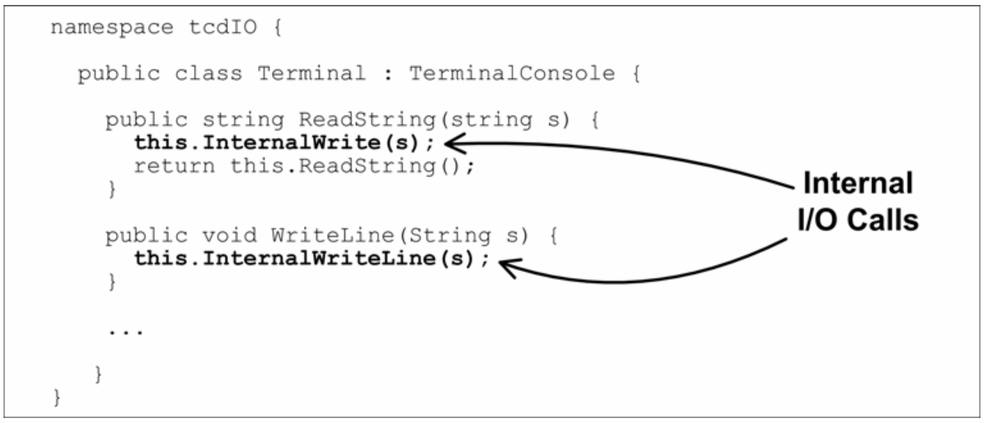

```
namespace tcdIO {

  public class Terminal : TerminalConsole {

    public string ReadString(string s) {
      this.InternalWrite(s);
      return this.ReadString();
    }

    public void WriteLine(String s) {
      this.InternalWriteLine(s);
    }

    ...

  }
}
```

Figure 2.7: Details of `class Terminal` implementation.

In contrast, context properties specified in terms of attributes are simpler to use and their presence provides feedback on refactored code. With attributes, fewer clerical issues exist in that it is clear from the placement of an attribute which point of code a context property influences. When reasoning about a component, the attribute provides an explicit indication of the presence of context properties. Moreover, it is possible to infer the property being applied from the name of the attribute. Based on the appearance of an attribute, we can determine if refactored code is consistent with the original version. For instance, the disappearance of an attribute from refactored code would alert the developer to a missing context property.

2.2 Canonical Aspect-Oriented Mechanisms

Although a number of AOP technologies exist, they draw their aspect-oriented characteristics from exploiting one or more of the five canonical aspect-oriented mechanisms. In support of so called *hybrid* approaches [Ras'01], in which a variety of aspect-oriented mechanisms are available, we should point out that they have been noted to improve separation of concerns. Indeed, prominent AOP tools such as AspectJ have been described as using multiple aspect-oriented mechanism [Mas'03]. However, we focus our discussion on the five key aspect-oriented mechanisms knowing that an understanding of these mechanisms will allow any AOP tool to be understood. To reiterate the introduction, these canonical mechanisms along with their archetype implementations are as follows:

- The pointcut-advice mechanism exemplified by the work of the AspectJ Team [Asp'00]
- Class composition exemplified by the work of the MDSOC project [IBM'00a]
- Object-graph traversal exemplified by the work of the Demeter Project [Lie'00]
- Open class composition, which originated with mixins [Moo'86], but is exemplified by the inter-type declaration semantics of AspectJ
- Composition Filters (CF) object model extensions available with tools such as ComposeJ [Car'01]

We can summarise each mechanism with an overview of how the mechanism allows an application to be crosscut. A pointcut-advice mechanism crosscuts the execution flow of an application. Method calls, method execution and field access can be selected and their semantics modified by specifying the execution of advice relative to selected join points. Class composition allows an application to be partitioned into declaratively complete subsystems that crosscut the types and type hierarchies of an application. Being declaratively complete, each subsystem will compile independently, which allows its development to go on independently from other subsystems. Open class composition allows type members and interface implementations to be added to a selection of existing classes without changing the source code of these classes. Specifically, the inheritance declarations used to define these classes are unchanged. Object-graph traversal mechanisms allow the succinct description of traversals that crosscut the object graph of an application. This avoids the need to embed traversal infrastructure in object graph elements, and it makes traversals flexible to changes in the object graph. A Composition Filters model provides crosscutting views of the messages passed between objects in an application. Although the CF model and context properties share a focus on message interception, the CF model provides a programming model with specialised syntax for defining filters that perform message manipulation. In contrast, context properties offer no new syntax and only a reflection API for examining messages.

The aspect-oriented mechanisms are supported as extensions to a variety of programming languages, but with these extensions the focus is on porting the aspect model to a programming language. Moreover, a strategy for language-independence as defined in literature [Laf'03] does not exist for every aspect-oriented mechanism. In the initial approaches to multilingual support, aspect-oriented extensions were developed on a language by language basis. For example, separate projects were undertaken to develop PA semantics for C++ (AspectC++ [Spi'04]), C (AspectC [Coa'01]), and Ruby (AspectR [Bry'02]). Likewise, the Demeter Project developed object traversal implementations for C++ [Lie'96] and then later for Java [Orl'04]. More recent work has examined a simplified means of multilingual support [Gra'04]. Rather than re-implementing a weaving engine, this approach allows the same weaver to be ported to different compilation tools by mapping weaving to language transformation systems, which are already available or easily implemented for the target language. Providing a plethora of extensions and a general porting mechanism does not address the language-independence problem which is to allow aspects and components developed in a variety of languages to be freely intermixed. However, strategies for language-independence have been proposed for both the PA model [Lam'02, Sch'02, Laf'03], and the CF model [Gar'03, Ber'04b], which are identified during our analysis of each technique.

In the following subsections we characterise the canonical aspect-oriented mechanisms. To put each mechanism in context, we summarise its historical background. The mechanism is introduced, and then its semantics are explained in terms of an aspect model. The specification of crosscutting is characterised in terms of whether the mechanism uses an approach closer to name-based or property-based crosscutting, and how this allows the mechanism to support obliviousness. The mechanism for reuse is identified along with strategies for language-independence. Finally, we summarise whether the mechanism is suitable for a replacement to aspect-based properties in terms of its ability to solve preplanning and tailorability problems with context properties and to provide a simple means of aspect reuse and language-independence.

2.2.1 Pointcut-Advice Model

2.2.1.1 Background

Pointcut-advice (PA) mechanisms originated with an effort to provide a generic aspect writing mechanism for ‘D’. ‘D’ was a language framework [Lop'97] that provided two aspect-specific languages for writing the synchronization and remote communications characteristics of distribute applications. Here, aspect-specific languages correspond to domain-specific languages in which the domain functionality is a crosscutting concern. In addition, ‘D’ included a PA mechanism, called JCore, to provide a general means for writing crosscutting functionality. Eventually, the aspect-specific languages were phased out in favour of a focus on PA mechanisms, and the tool renamed AspectJ [Kic'01b]. AspectJ now defines the de facto standard to which implementations of a PA mechanism are compared as occurs in [Mas'03].

2.2.1.2 Overview

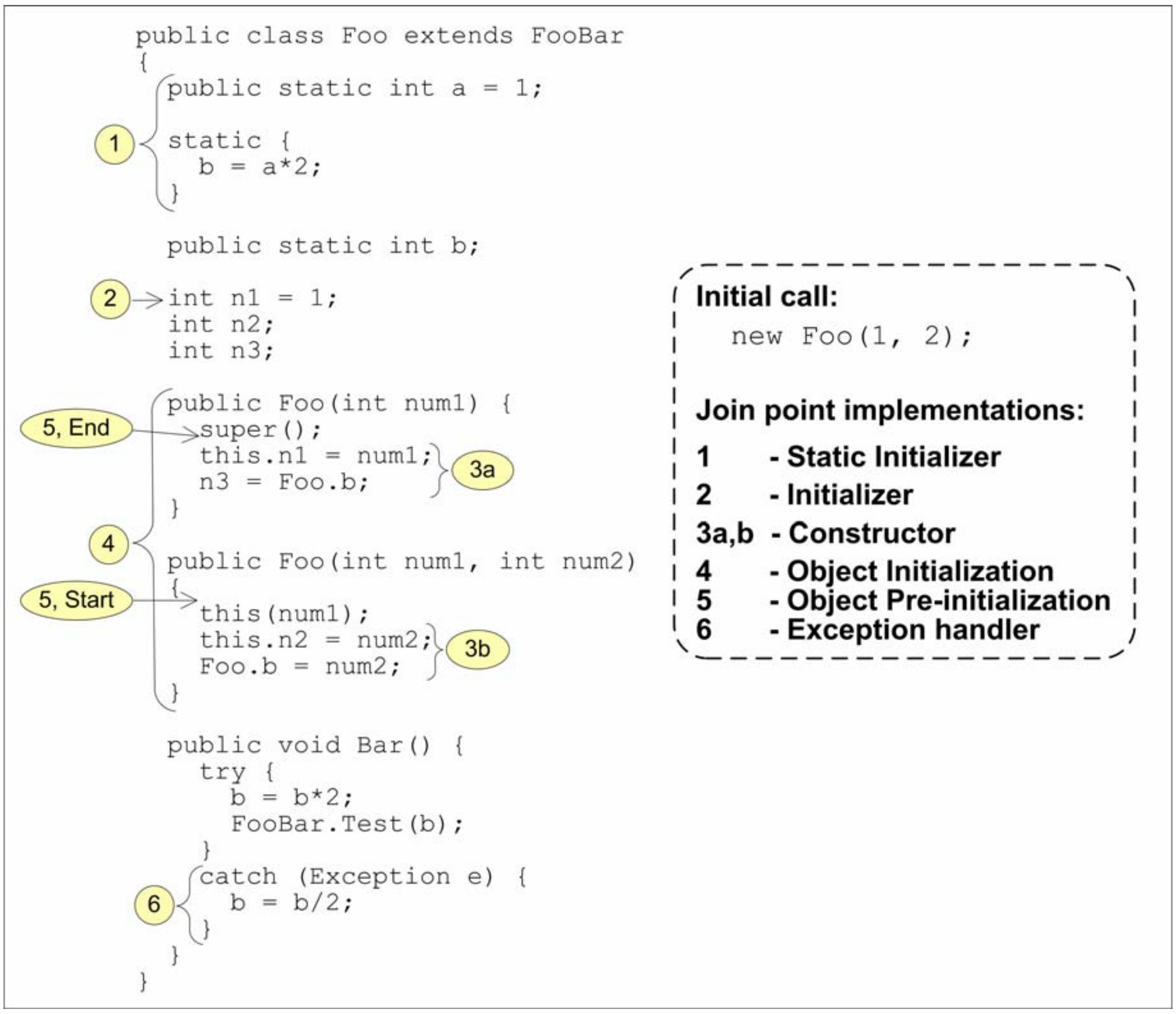
AspectJ provides language extensions to Java that allow the definition of aspect types [Kic'01b]. What distinguishes an aspect type in AspectJ from a regular Java class is the presence of pointcut specifications that identify sets of join points and advice specifications that influence the behaviour of join points in these sets. Pointcuts are specified in terms of primitive pointcut designators that are combined using Boolean logic. A variety of primitive pointcut designators are available for selecting “well-defined points in the execution flow of the program” [Kic'01a], referred to as dynamic join points in AspectJ terminology [Asp'02]. The specification of advice involves defining a method body and coupling its execution to the execution of join points in a pointcut. Advice can access the execution context of a join point through two means. A reflective API is available, but a more efficient approach is to bind join point execution parameters to typed formal parameters, which are accessible within the method body defined by advice.

2.2.1.3 Join Point Model

This section describes the join point model available with AspectJ V1.0.6 [Asp'02]. The AspectJ join point model [Asp'02] includes three categories of join point each consisting of several join point types, and these are summarised in Table 2.2. The categories correspond to method execution, client-side calls, and field accesses. The different join point types within the execution join point categories allow fine-grained distinctions between regular method execution and the execution of code bodies during class initialization, object initialization and exception handling. The fine-grained execution join point types are identified in Java code shown in Figure 2.8. The join points occur when the `new` operator is used to create a new object instance of `class Foo`. The first step to instantiating the object is to load the class. When a class is first loaded into the runtime environment, the static initializer executes. This static initializer corresponds to the execution of static assignments as well as class-static blocks of code, which are sometimes referred to as class constructors. This block of code is labelled with a '1' in the diagram. Next, an instance of `Foo` is created, and the initializer executes. Initializer execution corresponds to instance member assignments outside a method body. These are labelled in diagram as '2'. All these assignments are executed at the time of object construction, but their source code definition lies outside the body of the constructor. Constructor execution corresponds to the execution of the body of a single constructor method, but it does not include calls made to a super class constructor or another constructor of the same class. In Figure 2.8, the new operation invoked is `new Foo(1, 2)`, and so two constructor executions are required to initialise the object. These are labelled 3a and 3b. Object initialization encompasses the execution of all constructors and initializer code that occurs during object instantiation. In Figure 2.8 object initialization is the code identified by the label '4'. The code labelled '4' implicitly includes the initializer, as initializer code is embedded into the body of a constructor that calls a base class constructor. In contrast, object pre-initialization, corresponds to direct calls to constructors from the body of the first constructor called during object initialization. Such calls correspond to the use of `this()` and `super()` at the beginning of constructors. In Figure 2.8, we identify the start of this join point as the initial call to `this()` and the end as the return from `super()`. Although not involved in object creation, an example of a handler execution join point is identified in Figure 2.8 and labelled '6'. Handler execution join points correspond to blocks of code in the `catch` and `finally` blocks found in method bodies.

Table 2.2: Join point types distinguished in the AspectJ Pointcut-Advice aspect model.

Join point category	Join point types
Execution	Method execution Initializer execution Constructor execution Static initializer execution Handler execution Object initialization
Call	Method call Constructor call Object pre-initialization
Field access	Field reference Field assignment



```
public class Foo extends FooBar
{
  public static int a = 1;

  static {
    b = a*2;
  }

  public static int b;

  int n1 = 1;
  int n2;
  int n3;

  public Foo(int num1) {
    super();
    this.n1 = num1;
    n3 = Foo.b;
  }

  public Foo(int num1, int num2)
  {
    this(num1);
    this.n2 = num2;
    Foo.b = num2;
  }

  public void Bar() {
    try {
      b = b*2;
      FooBar.Test(b);
    }
    catch (Exception e) {
      b = b/2;
    }
  }
}
```

Initial call:

```
new Foo(1, 2);
```

Join point implementations:

1	- Static Initializer
2	- Initializer
3a,b	- Constructor
4	- Object Initialization
5	- Object Pre-initialization
6	- Exception handler

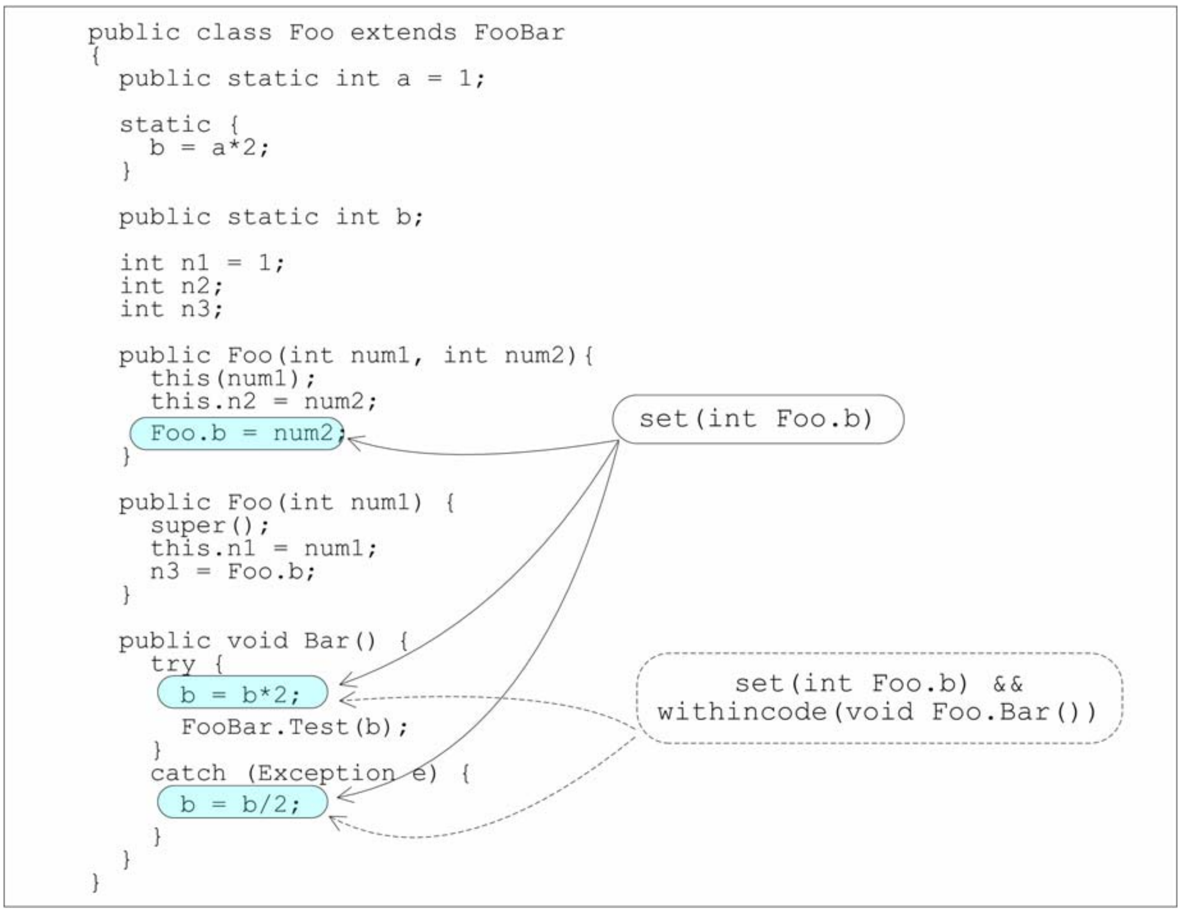
Figure 2.8: Java source code samples for select join point types presented in Table 2.2.

2.2.1.4 Means of Identifying Join Points

Of the three categories of the primitive pointcut designator that exist for the PA mechanism, Table 2.3 presents those that identify join points in terms of metadata descriptions. These designators identify join points according to the join point's implementation or the location of this implementation. To select a join point by implementation, a developer should pick a primitive pointcut designator corresponding to the join point type, and then provide as an argument the metadata description of the desired join point's implementation. Selecting join points by implementation location corresponds to the use of `within` and `withincode` designators. These select all join points, regardless of join point type, within a type or within a method's body, depending on which designator is used.

Table 2.3: Designators specified with a signature or type pattern.

Designator	**Joint points selected**
`call(`*`Signature`*`)`	Method and constructor calls.
`execution(`*`Signature`*`)`	Method and constructor execution.
`initialization(`*`Signature`*`)`	Object initializer execution.
`get(`*`Signature`*`)`	Field reference.
`set(`*`Signature`*`)`	Field assignment.
`Handler(`*`TypePattern`*`)`	Exception handler execution.
`staticinitialization(`*`TypePattern`*`)`	Static initializer execution.
`within(`*`TypePattern`*`)`	All join points defined by the selected type.
`Withincode(`*`Signature`*`)`	All join points defined within method or constructor matching declarations



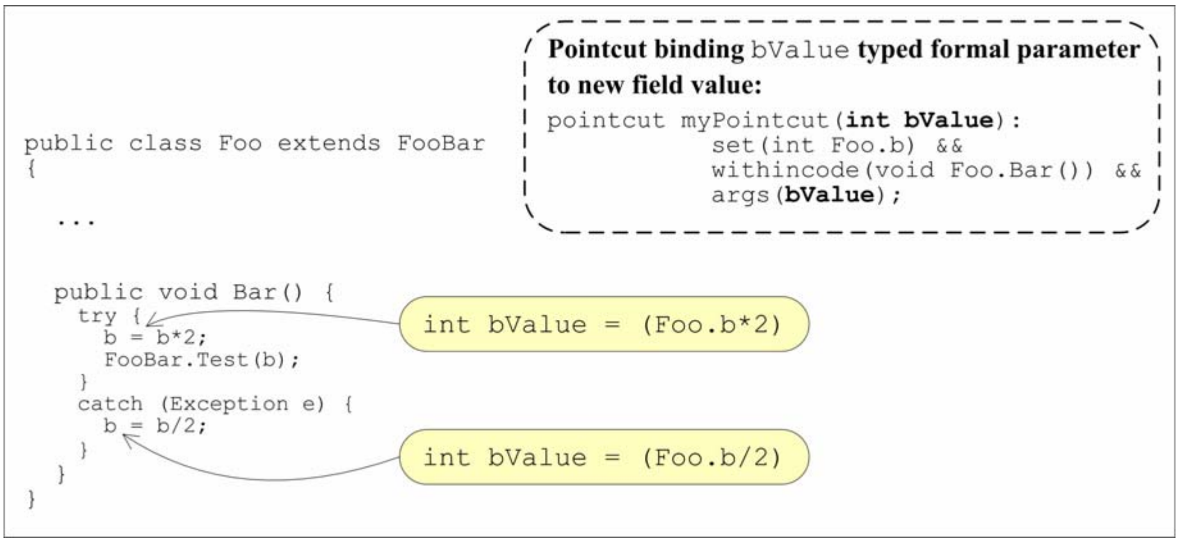
```
public class Foo extends FooBar
{
  public static int a = 1;

  static {
    b = a*2;
  }

  public static int b;

  int n1 = 1;
  int n2;
  int n3;

  public Foo(int num1, int num2){
    this(num1);
    this.n2 = num2;
    Foo.b = num2;
  }

  public Foo(int num1) {
    super();
    this.n1 = num1;
    n3 = Foo.b;
  }

  public void Bar() {
    try {
      b = b*2;
      FooBar.Test(b);
    }
    catch (Exception e) {
      b = b/2;
    }
  }
}
```

Figure 2.9: Example of logical combination of primitive pointcut designators from Table 2.3.

Table 2.4: Designators that can expose execution context with typed formal parameters.

Designator	Joint points selected
`this(`*`TypePattern or Id`*`)`	Join points in which the object bound to `this` is an instance of a particular type.
`target(`*`TypePattern or Id`*`)`	Join points in which the object on which a call or field operation is applied to is an instance of a particular type.
`args(`*`TypePattern or Id,`*` ...)`	Join points where there are arguments whose types match those listed by the designator.

Sets of join points can be refined by combining primitive pointcut designators. Intersections of join point sets correspond to the use of the logical *and*, while unions correspond to the use of the logical *or*. Sets can be negated and evaluation order managed. In the case of AspectJ, the syntax of logical *and* and logical *or* are '`&&`' and '`||`', while logical not is symbolised with a leading '`!`'. Round brackets are used to manage evaluation order. Figure 2.9 provides a simple example in which pointcuts are used to select join points from the `Foo` class that appeared in Figure 2.8. In the top pointcut, all field accesses to `Foo.b` are selected in which the value of `Foo.b` is set. This corresponds to the use of the primitive pointcut designator `set`. In the bottom pointcut, the `withincode` primitive pointcut designator is combined with the previous pointcut using '`&&`' to narrow the set of join points selected to those that appear in the `Bar` method.



```
public class Foo extends FooBar
{

  ...

  public void Bar() {
    try {
      b = b*2;
      FooBar.Test(b);
    }
    catch (Exception e) {
      b = b/2;
    }
  }
}
```

Pointcut binding `bValue` typed formal parameter to new field value:

```
pointcut myPointcut(int bValue):
            set(int Foo.b) &&
            withincode(void Foo.Bar()) &&
            args(bValue);
```

```
int bValue = (Foo.b*2)
```

```
int bValue = (Foo.b/2)
```

Figure 2.10: Sample use of typed formal parameter in a pointcut.

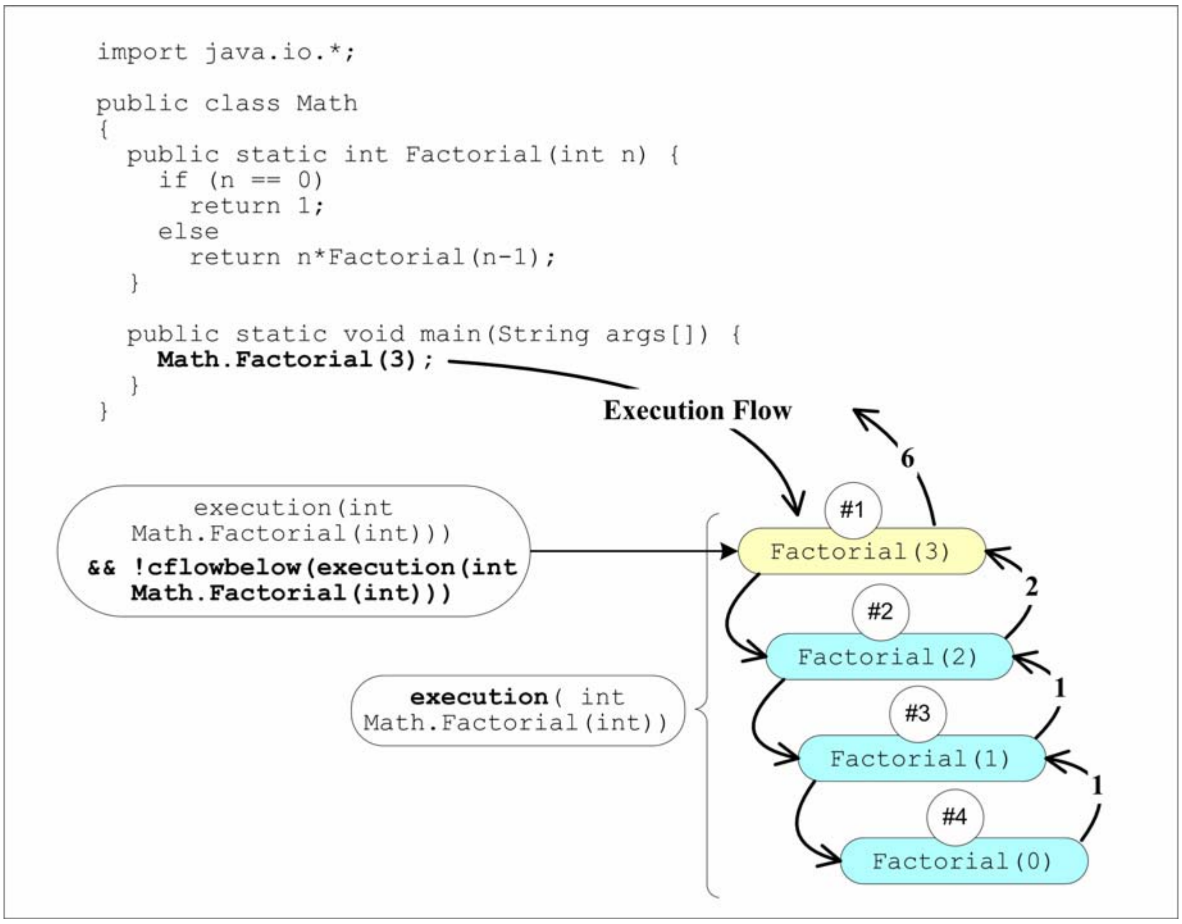
Access to join point execution context can be obtained through typed formal parameters or a reflective API. *Typed formal parameters* [Asp'02] are variables declared in a pointcut and bound to join point execution context using the primitive pointcut designators shown in Table 2.4. These primitive pointcut designators match join points depending on the type of parameters in the join point execution context. `this` matches the type of the object executing the join point. `target` matches the type of the object used to reference the join point's implementation, and `args` matches the join point execution parameters. The parameters of these primitive pointcut designators are types. These types can be specified explicitly by using the type name. Alternatively, type can be specified implicitly by providing the name of a typed formal parameter whose declared type is then used as the parameter. In this case, the typed formal parameter is assigned a reference to the parameter it matches. As we will see, this typed formal parameter can then be used by advice that executes at the join point. Different implementations of the PA model make available a reflective API from which join point execution context can also be obtained. However, the need to set up the reflective description of a join point introduces significant execution time overhead, and so typed formal parameters are preferable. Figure 2.10 provides an example use of the typed formal parameters. This example revises the bottom pointcut of Figure 2.9 to include a typed formal parameter that is bound to the new value that will be assigned to the `b` variable in the `Bar` method. The diagram shows how the value referenced by `bValue` varies depending on the join point executing.

Table 2.5: Designators specified with a pointcut.

Designator	**Joint points selected**
`cflow(`*`pointcut`*`)`	All join points encountered during the execution of join points identified by the pointcut.
`Cflowbelow(`*`pointcut`*`)`	Identical to *`cflow`*, but does not include the join points identified by the pointcut argument.

The designators of Table 2.5 allow the selection of join points to take into account execution flow. Selecting join points based solely on a metadata description of their implementation is unsuitable when code is re-entrant. Take the example of recursion in functional programming, in which methods use recursion instead of loops. The recursive invocations are called with intermediate parameters and return intermediate results, and so they may not be of as much interest as the initial method invocation and its result. For example, in Figure 2.11 `class Math` implements the `Factorial` function using recursion. The invocation `Factorial(3)` in the `main` method causes a series of recursive calls. The initial call is shown at the bottom of the diagram and labelled `#1`, whilst subsequent recursive calls are labelled `#2`, `#3`, and `#4`. The recursive calls are said to be in the execution flow of `Factorial(3)`, as these method executions are required in order for the execution of `Factorial(3)` to complete. The `cflow` and `cflowbelow` primitive pointcut designators allow pointcuts to take into account execution flow. `cflow` broadens a pointcut specification by adding all join points that appear in the execution flow of join points in the pointcut. In contrast, `cflowbelow` selects join points that appear in the execution flow of join points of another pointcut. For example, in Figure 2.11 the pointcut `execution(int`

`Math.Factorial(int))` selects each of the method executions that result from a call to `Factorial`. For the call `Factorial(3)` in Figure 2.11, four execution join points match, but only the first invocation will return the final result. To capture only this join point, `cflowbelow` can be used to remove the intermediate method executions from our pointcut. `cflowbelow(execution(int Math.Factorial(int)))` selects the unwanted join points, which are then removed from the set of selected join points using the logical operations pictured in Figure 2.11.



```
import java.io.*;

public class Math
{
  public static int Factorial(int n) {
    if (n == 0)
      return 1;
    else
      return n*Factorial(n-1);
  }

  public static void main(String args[]) {
    Math.Factorial(3);
  }
}
```

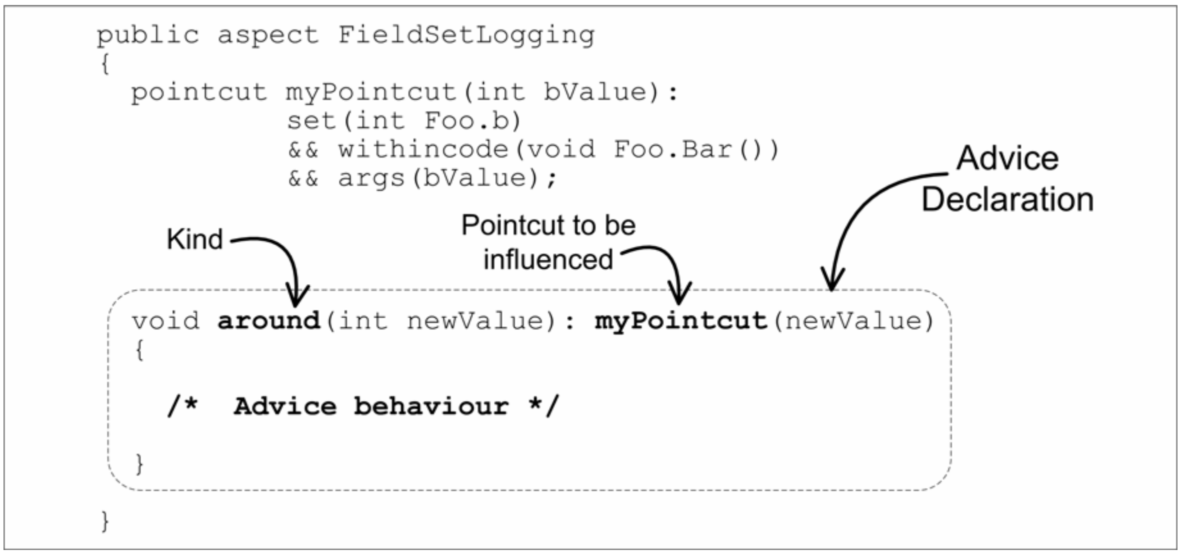
Execution Flow
execution(int Math.Factorial(int))) && !cflowbelow(execution(int Math.Factorial(int)))
execution(int Math.Factorial(int))
#1
Factorial(3)
#2
Factorial(2)
#3
Factorial(1)
#4
Factorial(0)
6
2
1
1

Figure 2.11: Example of execution flow-based join point selection with `cflow`.

PA models support both name-based crosscutting and property-based crosscutting by varying the signatures and type pattern arguments used to parameterise primitive pointcut designators. Name-based crosscutting corresponds to the literal expression of signatures and type patterns. With name-based crosscutting, the signatures and type patterns used in a pointcut must match the implementation of the targeted join points exactly. This name-based crosscutting supports obliviousness as the specification of join points is noninvasive. So long as the declarations in the component being crosscut are implemented to match existing name-based crosscutting, the component developer can be unaware of the application of an aspect. In property-based crosscutting, the signatures and type patterns used in a pointcut are only partially specified. This is achieved by allowing declarations to be written in terms of regular expressions. For example, rather than writing a method name explicitly, the regular expression "`Write*`" is used to select method names with the prefix "`Write`". More recently, attributes have been explored as a means of property-based crosscutting [Mas'03] in tools such as AspectWerkz [Bon'04b]. In this alternative form of property-based crosscutting, an attribute can be specified instead of a partial signature or type name. Type or type member declarations sporting the specified attribute are selected according to whether the primitive pointcut designator is expecting a type pattern or type member signature. In the case that property-based crosscutting uses signatures, obliviousness can be achieved. However, the use of attributes leaves artefacts in the component corresponding to the aspect and nothing else. Being invasive, the use of attributes does not strictly adhere to obliviousness.

2.2.1.5 Means of Modifying Join Point Semantics

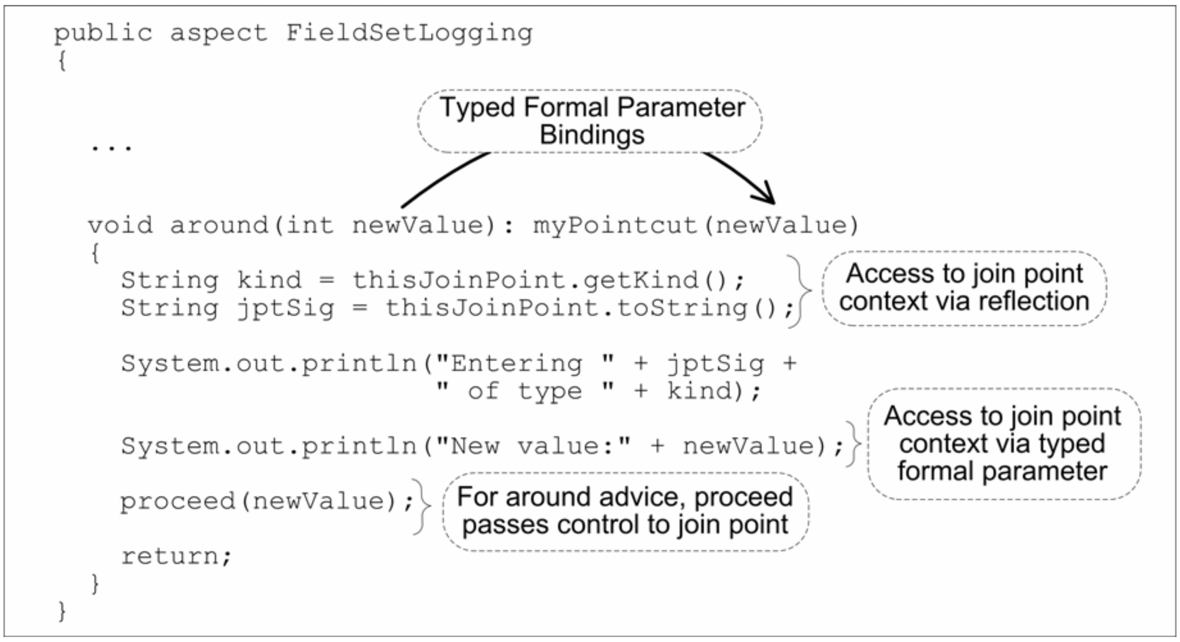
Advice is a specification for behaviour executed relative to join points in a pointcut. A sample advice declaration written in AspectJ syntax appears in Figure 2.12. This advice declaration first specifies its kind. Kind indicates when advice behaviour is executed relative to affected join points. In the case of Figure 2.12, advice will be executed around existing join points, which is to say that advice behaviour is executed instead of the join



```
public aspect FieldSetLogging
{
  pointcut myPointcut(int bValue):
           set(int Foo.b)
           && withincode(void Foo.Bar())
           && args(bValue);

  void around(int newValue): myPointcut(newValue)
  {

    /*  Advice behaviour */

  }
}
```

Figure 2.12: Example advice declaration.

point and that the join point can then be called from within the advice behaviour. As mentioned earlier, a PA model makes available before and after advice. Before advice executes before the join point, while after advice executes afterwards. Since a join point can either complete normally or throw an exception, there are three types of after advice. *After returning* advice executes when a join point executes normally, while *after throwing* advice executes when a join point throws an exception. Finally, *after* advice executes in both cases. Figure 2.12 also points out the use of pointcuts in advice. A pointcut is used to select the set of join points whose semantics will be modified. In this example, the pointcut is selected by citing the name of an existing pointcut declaration. Finally, advice must specify the behaviour that is executed relative to join points. In the AspectJ implementation, the behaviour executed is specified in the code block that follows the advice declaration. An example of such a block is shown in Figure 2.13, which highlights the two means by which advice accesses join point state. The figure provides examples of



```
public aspect FieldSetLogging
{

  ...

  void around(int newValue): myPointcut(newValue)
  {
    String kind = thisJoinPoint.getKind();
    String jptSig = thisJoinPoint.toString();

    System.out.println("Entering " + jptSig +
                       " of type " + kind);

    System.out.println("New value:" + newValue);

    proceed(newValue);

    return;
  }
}
```

Typed Formal Parameter Bindings
Access to join point context via reflection
Access to join point context via typed formal parameter
For around advice, proceed passes control to join point

Figure 2.13: Example specification around advice written in AspectJ.

join point state being accessed through typed formal parameters declared in the pointcut as well as through a reflective API. In the case of AspectJ, typed formal parameters are bound to variables in the scope of the advice behaviour code block. So, the `bValue` typed formal parameter declared in the `myPointcut` pointcut of Figure 2.12 is bound to the `newValue` variable in Figure 2.13. `newValue` then has access to the value being assigned by join points in the `myPointcut` pointcut. The alternative to typed formal parameters is to use a reflective API to introspect on join point state. For example, AspectJ makes available keywords such as `thisJoinPoint` that give access to reflective objects describing the join point at which advice behaviour is executing.

2.2.1.6 Analysis

A PA mechanism does not have the preplanning and tailorability problems of contextual composition, nor does it preclude attribute-based property selection or language-independence. The aspects available with a PA mechanism are well suited to solving the tailorability problem with context properties. PA mechanisms are consistent with contextual composition in that both allow the execution flow of a program to be influenced. Also, there are examples of common context properties such as persistence [Soa'02, Ras'03] being addressed with PA mechanisms. While the preferred strategy for reusing properties implemented with these aspects is the use of abstract aspects [Pic'03, Ras'03], even where attribute-based property-based crosscutting is supported [Bon'04a]. Abstract aspects define advice and leave the pointcut partly unspecified, and reuse involves implementing the pointcuts, and thus aspect modification. This approach is unsatisfactory, because the crosscutting semantics of the aspect are being revised. We can see an alternative consistent with contextual composition in which attributes are used for property-based crosscutting. This alternative involves using attributes in place of type, method signatures and field signatures in pointcut specifications. While such an approach offers the same attribute-based property selection offered by context properties, there is no programming model for separately expressing aspect bindings and aspect crosscutting semantics written with attributes. Finally, introducing PA mechanisms need not place any restrictions on OO programming languages [Kic'01b], and we see no evidence of restrictions being imposed on other languages types; however, the interoperability of aspects and components written in a variety of languages not addressed by the programming model.

2.2.2 Class Composition in MDSOC

2.2.2.1 Background

Multi-dimensional separation of concerns (MDSOC) [IBM'00a, Oss'01c] is a software evolution technology that applies class composition to the separation of concerns in a software application. MDSOC provides the means to partition an application according to a particular dimension of concern. A concern [Oss'01b] is an arbitrary engineering issue such as software features. The partitions are referred to as hyperslices, and class composition is used to facilitate the development of software in terms of hyperslices. A hyperslice [Tar'99] is a declaratively complete set of partial types that corresponds to a concern. The hyperslice is declaratively complete in that it will independently compile allowing independent development. The set of types is partial in that compilation is dependent on units, such as types or type members that are declared in the hyperslice, but may not all be defined within that hyperslice. Class composition is called upon to combine hyperslices so that all declared units have a concrete definition, and so that there are no conflicts when different hyperslices inadvertently define units with the same name.

2.2.2.2 Overview

The class composition mechanism used in MDSOC originated with research on subject-oriented programming (SOP) [Har'93, Mat'96, IBM'00b]. SOP aimed to provide a means whereby the direct specification of object classes could be avoided. Instead, classes emerge from the different roles that their objects play in subsystems of an application. SOP research points out that the intrinsic properties of an object that are of interest to an application are those exploited during object interactions, and so these interactions should be the basis of an object's definition. Interestingly, the definitions of the various intrinsic properties of an object may be at odds depending on the context in which an object is used. Thus, "the essential characteristic of subject-oriented programming is that different subjects can separately define and operate upon shared objects, without any subject needing to know the details associated with those objects by other subjects." [Har'93] To achieve partitioning, classes defining objects are themselves composed from a set of partial types for which there is one for each subsystem in which a class object plays a role. Such subsystems were referred to as *subjects* [Har'93]. In SOP, the implementation of a subject consists of several partial types classes that collaborate to achieve a particular functionality. Thus, subjects correspond to hyperslices in MDSOC [Tar'99]. An application is built from the composition of several subjects, and objects of the application are instantiated from classes that compose the object's functionality in each of the subjects in which it plays a role [Oss'94].

An example for motivating class composition in subject-oriented programming is the task of modelling a tree in software, where the tree has quite different properties depending on the context, or subject, in which it is used [Har'93]. If the subject were forest ecology, important characteristics of a tree would be the water consumption, the kind of habitat the tree provides and the age of the tree, while important operations might be shedding/growing leaves, growing higher, or growing fruit. If the same tree were considered for the subject of logging, somewhat different properties and behaviour would be of interest. For instance, the tree's wood type, height, diameter and value would be key characteristics, while important operations would be to it cut down and remove branches. Analogously, an application can consist of multiple subjects, each of which describes the interaction of overlapping sets of objects.

To allow applications to be developed in terms of subjects, subject-oriented composition provides two general categories of composition rules to allow subjects to be merged with minimal restrictions on their implementation. *Correspondence* rules [Oss'95] identify methods, fields and class names from different subjects that refer to the same entity. This allows a class to be developed in different subjects without the need to decide upon a common name across all subjects. The implication is that subjects can be developed by different development teams and possibly as separate applications. Recall the example of a tree object involved in ecology and logging. At some point an integrated forest management system may be developed that models both the ecological and logging characteristics of a tree. Rather than creating a new application, it would be simpler to integrate the existing ecology and logging applications. However, merging the two separately developed systems will likely fail due to naming discrepancies. For example, the tree class may be named differently in the two subjects mentioned. Correspondence rules over come such issues by providing the ability to map classes from one subsystem to another.

While correspondence rules deal with differing names, combination [Oss'95] rules deal with differing implementations. Subject-oriented composition offers *combination* rules to deal with conflicts arising from the presence of multiple definitions for methods and fields in subjects that are being composed. Of greatest concern is the meaning of a self-reference variable, e.g. `this`. Combination rules can define whether composed operations use `this` to access an object defined by the pre-composition class definition or that of the newly composed class. Of secondary importance is the resolution of variables that have the same name prior to composition. In both cases, the combination rules dictate the meaning of variables either on a class basis or on a method basis.

2.2.2.3 Join Point Model

Class composition, as practiced with MDSOC, manipulates units. A *unit* is a syntactic construct in a particular language [Oss'01b], and when applied to object-oriented languages a unit corresponds to types and type members. For example, the class `Rational` in Figure 2.14 and `BetterRational` in Figure 2.15 define types, contain fields and contain methods. The types defined as well as their field and methods all correspond to units.

```
using System;
using System.Threading;

public class Rational {
  protected int numer, denom;

  public Rational(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
  }

  public Rational Add(Rational other) {
    int numerator = this.numer + other.numer;
    int denomenator = this.denom * other.denom;

    return new Rational(numerator, denomenator);
  }

  public override string ToString() {
    string info = numer.ToString() + " / "
                + denom.ToString();
    return info;
  }
}
```

Figure 2.14: Class modeling rational numbers.

```
using System;
using System.Threading;

public class BetterRational : Rational {
  public BetterRational(int numer, int denom): base(numer, denom){}

  public Rational Sub(BetterRational other) {
    int numerator = this.numer - other.numer;
    int denomenator = this.denom * other.denom;

    return new BetterRational(numerator, denomenator);
  }
}
```

Figure 2.15: Subtype of `class Rational` with an added mathematical operation.

2.2.2.4 Means of Identifying Join Points

With MDSOC, a concern describes a set of units, and a specification for a set of types and type members is referred to as a concern map. Concern mappings are able to select class members across multiple classes, regardless of the type hierarchy of those classes, and concern mappings are used to define hyperslices. In an example analogous to feature slicing shown in [Oss'01b], we could define hyperslices to separate the mathematical operations, data storage facilities, and display operation of classes `Rational` and `BetterRational`. The concern mappings to do this are shown in Figure 2.16, which assume that class `Rational` and `BetterRational` are defined in the `myRational` package. This concern mapping will create three hyperslices corresponding to different features. The `Data` feature hyperslice will have all classes, less the definitions for the `Add`, `Sub`, and `ToString` methods. The `Math` feature will have two abstract types corresponding to `Rational` and `BetterRational` with `Add` and `Sub` operations defined in the appropriate class and abstract declarations for the data and methods upon which `Add` and `Sub` are dependent. Similarly, the `Display` feature will have a single abstract type containing the `ToString` method.

```
package myRational: Feature.Data
operation Add: Feature.Math
operation Sub: Feature.Math
operation ToString: Feature.Display
```

Figure 2.16: Concern mapping that divides the class `Rational` in Figure 2.14 and `BetterRational` in Figure 2.15 along feature boundaries.

2.2.2.5 Means of Modifying Join Point Semantics

The modification of join point behaviour takes place via a hypermodule specification that specifies a set of hyperslices and their relationships. The relationships compose hyperslices by resolving abstract units in one slice with concrete definitions elsewhere and by resolving conflicts when multiple concrete definitions satisfy abstract declarations. These composition rules are based on those available with subject-oriented programming [Oss'01b]. Although relationships primarily offer structural composition, the behaviour of the composed program can be influenced by varying the meaning of members referenced by the code of a particular hyperslice. For instance, references to methods being called can be varied to allow different or multiple methods to be executed. In a simple example of composition, we can combine the data store and features of the Figure 2.16 concern map as show in Figure 2.17. In this figure, `mergeByname` is a simple relationship that states that units in different hyperslices with the same name correspond. It is simple in that it does not resolve conflicts that occur as a result of multiple definitions of concrete methods and concrete fields.

```
hypermodule Rationals_NoMath
  hyperslices: Feature.Data, Feature.Display
  relationship: mergeByName
```

Figure 2.17: Hypermodule constructed from concerns defined in Figure 2.16.

2.2.2.6 Analysis

The strengths of class composition are its symmetric composition model and its suitability for structural composition; however, these strengths do not make class composition well suited to replacing contextual composition. A class composition mechanism, as typified by the Hyper/J MDSOC technology [Oss'01c, Oss'01a, Oss'01b], lacks the aspect-component split central to initial AOP research. This led MDSOC researchers to note that a “key difference between MDSOC with Hyper/J and AOP as described in the literature [Kic'97] and exemplified by AspectJ [Kic'01b], is that AspectJ supports augmentation of a single model, whereas Hyper/J supports integration of multiple models." [Oss'01c] Identified in this contrast is the pointcut-based mechanism of AspectJ, which is said to offer a single model of an application in that it allows the manipulation of an entire program. That is, pointcut-advice specifications pick join points from the entire application. In contrast, MDSOC allows the manipulation of partial pieces of a program. Composition relationships used to define new hypermodules pick units from hyperslices, which provide a limited view of the entire application. The creation of a hyperslice makes sense if the concern being manipulated crosscuts the components of a system. However, with contextual composition, it is well encapsulated component functionality that is being manipulated. For this reason, an asymmetric composition paradigm suits. Also, MDSOC comes from a background of structural composition, whereas the method interception of contextual composition suits behavioural composition.

2.2.3 Aspect-Oriented Open Class Composition

2.2.3.1 Background

The precursor to open class composition is the concept of a mixin, which provides a means of making common features reusable across multiple class definitions [Cza'00]. Mixins originate from work on the Flavors programming system [Moo'86], where it was observed that the same facet of class behaviour is often repeated throughout an application. Duplicating the same code can be avoided with a mechanism that allows this functionality to be written once and applied multiple times. Originally, this mixing was achieved using inheritance and, in particular, multiple inheritance [Moo'86], as there will be cases when a class will want to take advantage of multiple mixins. Take the example of a class wishing to participate in a doubly linked list. To do so it may select a mixin that contains the methods required to add and remove elements from the list as well as the data members required to reference the next and previous list elements. In addition, it may be a requirement that that same class have the ability to participate in a hash table. In this case, a mixin for hash generation functionality would also be useful. However, the use of mixins is not to be confused with inheritance in general, which has other applications such as specialising an existing class or making abstract behaviour concrete.

2.2.3.2 Overview

Unlike mixins, open class composition allows noninvasive changes to a class definition from other classes in the application. The principle of open class composition is derived from elements of the multimethod functionality developed for the Dubious language [Mil'99]. Among their features, multimethods allow an object to define new methods for its constituent, or aggregated, objects in an idiom called *open objects* [Mil'99]. A focus on composing methods with objects makes sense in Dubious as it is a class-less OO language. In a class-based OO language, open object functionality is realised as open class functionality. "Open classes allow one to add to the set of methods that an existing class supports without creating distinct subclasses or editing existing code." [Cli'00] Open class composition mechanisms are consistent with the goal of mixins in that new operations are added to the class. However, a big difference is that open class composition is achieved without using the inheritance mechanism. Moreover, mixins encapsulate the additive behaviour into a single class, whereas open class composition does not focus on mechanisms for grouping additive behaviour into units of encapsulation.

Open classes can be applied as an aspect-oriented mechanism when the behaviour being added to objects is modeled as an aspect. This involves grouping the members to be injected into application classes into a single unit. This encapsulated behaviour can then be made to crosscut the objects of an application by identifying the types to which it will be applied in a central location in a noninvasive fashion. This is in contrast to the traditional approach taken with mixins in which behaviour being added was well encapsulated, but in which the classes wishing to avail of mixin behaviour had to annotate themselves with the appropriate inheritance declaration. An example of using open class composition as an aspect-oriented mechanism is in the centralization of infrastructure required to support traversals in the visitor pattern [Gam'94], as discussed in [Cli'00]. With the visitor pattern, each object type in the graph being traversed must implement a method to invoke the method of the visiting object appropriate to the object type being visited. Open classes allow the definition of a visitor object to include infrastructure requirements for object types being visited. In this example, open classes are crosscutting the types within an application.

Although both open class composition and MDSOC-style class composition provide mechanisms for combining classes, their composition goals differ in that open class composition adds behaviour to classes, whereas class composition in MDSOC adds classes to behaviours. Open class composition has a focus on making additive changes. It does not require combination and correspondence rules as existing functionality is not being replaced or remapped. Moreover, composition in MDSOC is interested in forming classes from roles of objects within subsystems that are described with *hyperslices*, while open class composition is interested in augmenting the functionality of a particular class definition.

2.2.3.3 Aspect Model

An aspect model for open class composition is realised in both MultiJava [Cli'00] and AspectJ [Asp'00], but of these two examples AspectJ best supports open class composition in the AOP sense. Each tool has as its join point model the types within a system. Since types are often used to express the static structure of a program, it is not surprising that these join points are sometimes referred to as *static join points* [Asp'02]. The means of identifying types is usually by name with the name specified either exactly, as in the case of MultiJava [Cli'00], or by regular expressions describing the type name as in AspectJ [Asp'00]. The means of modifying the semantics of the types identified is to add to the list of operations available, to the set of data members in a particular type implementation, and possibly to the list of types implemented or inherited from. In doing so, open class composition allows MultiJava [Cli'00] and AspectJ [Asp'00] to apply operations to crosscutting views for data that would otherwise be blocked by the data hiding properties of a class boundary. However, there remains an important distinction between AspectJ and MultiJava. AspectJ offers open class composition, referred to *introduction* [Asp'02], or more recently as *inter-type declarations*, as a constituent of an aspect type. In contrast, the grammar of MultiJava [Cli'00, Gos'00] offers no means of encapsulating open class composition statements. An aspect abstraction is not a required element of an aspect model; however, it is the purpose of AOP to provide logical encapsulation of crosscutting functionality. This makes the approach to open class composition offered by AspectJ preferable to that available with MultiJava [Cli'00].

Figure 2.18 demonstrates open class composition-based crosscutting with a sample of AspectJ source used to insert doubly linked list functionality into a `class Foo`. In the `DblLinkListAspect` declaration, the `declare parents` statements adds the `DblLinkList` interface to the list of types implemented by class `Foo`. Next, elements whose identifiers are prefixed with by type `Foo` are inserted into the `class Foo` declaration. The specification of types to which new functionality is introduced is consistent with the approach used in AspectJ to specify PA model pointcuts. Thus, the characteristics of name-based and property-based crosscutting for class composition are consistent with those of the PA model implementation discussed in section 2.2.1.

```
public interface DblLinkList {
  public void Append(DblLinkList newObj);
  public void Insert(DblLinkList newObj);
  public void Remove(DblLinkList oldObj);
}
```

```
public aspect DblLinkListAspect {

  declare parents: Foo implements DblLinkList;

  public DblLinkList Foo.Next;
  Public DblLinkList Foo.Prev;

  public void Foo.Append(DblLinkList newObj) {}
  public void Foo.Insert(DblLinkList newObj) {}
  public void Foo.Remove(DblLinkList oldObj) {}
}
```

Figure 2.18: AspectJ source for inserting doubly linked list functionality into a `class Foo`.

2.2.3.4 Analysis

The difficulty with using open classes to replace contextual composition is that open classes make purely additive structural changes. The point of open class composition is not to influence existing methods, as the changes being made to classes do not influence existing behaviour. In contrast, contextual composition is not meant to add new methods, but instead attaches additional behaviour to execution of existing methods.

2.2.4 Object Graph Traversal

2.2.4.1 Background

Work by the Demeter project [Lie'00] focuses on expressing object graph traversals that crosscut the object hierarchies of an application. In an *object graph* [Lie'04], objects form the nodes, and references between objects form the graph's arcs. An aspect-oriented means to address object graph traversal emerged from problems with maintaining code that relies on graph traversals. Implementing a traversal requires a method to access the object references between objects. In its simplest form, these references are public data. However, using this data directly involves making explicit assumptions about the chain of references from an object accessing the method to the methods being invoked. When the invoking method includes direct accesses to each data member storing an object reference, details of the object graph become built into its implementation. This leads to brittle code that will break when the object graph is modified. The *Law of Demeter* [Lie'96] proposes programmers implement traversals with methods that do not make assumptions about the object graph. Such methods should not directly access the data members containing object references when those data members are contained in objects of another class. Rather, object references should be accessed via methods. Doing so avoids having methods of one class dependent on the data of objects defined by another. Following the Law of Demeter leads to a second approach to building an object traversal, which involves using intermediate methods whose only role is to route messages between objects that do not have direct references to each other [Orl'01]. However, this too leads to maintenance difficulties. Revising the object graph requires changes to the methods responsible for providing the routing.

2.2.4.2 Overview

In response to the short comings of OO based techniques, the Demeter project proposes adaptive programming (AP) [Lie'96] as an alternative for writing behaviour that crosscuts the object graph of an application. AP makes available propagation patterns (now called adaptive methods [Lie'01]) that consist of a traversal strategy and an adaptive visitor. Here, a *traversal strategy* is “a high-level description of how to reach the participants of the computation” [Lie'01], while an *adaptive visitor* specifies “what to do when each participant has been reached” [Lie'01]. The approach is adaptive, because adaptive methods will organise themselves around a program’s object graph, allowing the object graph to change without the need to update operations that are dependent on traversing the graph. Adaptive programming is similar to the application of the visitor pattern; however, AP is not as invasive as the visitor pattern. For instance, AP avoids the need for the node being visited to provide a method to activate visitor behaviour. Moreover, AP avoids the need for a programmer to manually add traversal infrastructure to an application’s objects.

2.2.4.3 Join Point Model

Adaptive programming has for sometime been recognised as an aspect-oriented programming mechanism [Elr'01, Lie'04] in which aspects correspond to adaptive methods [Lie'01]. In adaptive programming, the join point model used consists of the set of object visitations possible in an application. By using the term visitation, we are referring to the traversal path to an object plus the execution of methods when the object is reached.

2.2.4.4 Means of Identifying Join Points

Join point selection is via an object graph traversal specification that selects a particular set of visitations based on the traversal strategy specified and the starting point object for the traversal. To clarify, we refer to Figure 2.19, which contains source for a simple aspect method written using the DJ tool [Orl'01]. The example traversal strategy is specified with the string `"from Company to Salary"`, which selects visitations to objects of type

Salary. Whether these objects are Salary class instances, or simply referenced by variables of type Salary is an implementation detail. Note that since object graphs are typically directed, the starting point of a traversal places considerable restrictions on which objects can be reached. In Figure 2.19, the starting point is an object implementing the Company type. In an application that calculates salaries for multiple companies, having a single starting point will limit any given traversal to only salary objects that correspond to a particular company. The visitations are no longer brittle to changes in the object graph, because their implementation does not include object graph navigation code. Instead, the infrastructure required to navigate the object graph is created automatically by the weaver.

```
import edu.neu.ccs.demeter.dj.ClassGraph;
import edu.neu.ccs.demeter.dj.Visitor;

class Company {
  static ClassGraph cg = new ClassGraph () ; // class structure
  Double sumSalaries () {

    String s = "from Company to Salary" ; // traversal strategy

    Visitor v = new Visitor ( ) { // adaptive visitor
      private double sum;
      public void start ( ) { sum = 0.0 };
      public void before (Salary host) { sum += host.getValue() ; }
      public Object getReturnValue ( ) { return new Double (sum) ; }
    };

    return (Double) cg. traverse (this, s, v) ;
  }
  // . . . rest of Company definition . . .
}
```

Figure 2.19: Simple adaptive method taken from [Orl'01].

Adaptive programming adheres to the obliviousness principles of AOP with a strictly property-based approach to crosscutting. Name-based crosscutting of object graph traversals defeats the purpose of AP, which is to avoid coupling traversals with implementation of an application's object graph. Instead, AP provides strong semantics for specifying traversals in terms of properties. The objects being traversed contain no artefacts corresponding to traversal specifications, and so the property-based crosscutting provided supports obliviousness.

2.2.4.5 Means of Modifying Join Point Semantics

The modification of join point semantics involves attaching execution behaviour to the visitation. The behaviour of a visitation is written in terms of the adaptive visitor. In Figure 2.19, this visitor corresponds to an anonymous class instance assigned to a variable of type Visitor. Being a class instance, an adaptive visitor can contain its own state. In Figure 2.19, the visitation behaviour is defined in terms of functions whose name determines when during a traversal the method is executed. For example, the method `before` is executed upon arrival at the destination node, and access to the node's interface is via a method argument.

2.2.4.6 Analysis

Object graph traversal is useful for allowing the object graph to be varied; however, the ability to revise existing behaviour does not exist. Thus, object graph traversal is not a suitable replacement for contextual composition.

2.2.5 Superimposition of Composition Filters

2.2.5.1 Background

The purpose of the Composition Filters (CF) model [Aks'92, Ber'94] is to avoid the tangling that occurs when database services such as transactions and persistence are orchestrated for an object. Specifically, it was observed that such orchestration involved "using database services through embedded data manipulation statements." [Aks'92] Research motivating CF identified the need to avoid this tangling without interfering with the ability of object-oriented mechanisms to address other program functionality. For example, relying on inheritance to compose persistence with objects in a single inheritance language impedes the use of inheritance for accessing other services. The solution in the CF model is to provide an extension to an object-oriented programming model that allows the manipulation of messages incoming to and outgoing from an object on an object-wide basis [Ber'04a].

2.2.5.2 Overview

The CF model supplants the supremacy of classes as a means of defining object types in OO languages with a new abstraction called a *concern* [Car'01]. The principle contribution of a concern is the introduction of abstractions for writing reusable message manipulation logic. Message manipulation is written in terms of reusable *filter modules,* which consist of filters with supporting methods and object instances. Broadly speaking, filters examine incoming messages, and they take a specific action for messages that match the conditions of the filter. *Superimposition* [Car'01] allows a concern to apply filter modules to arbitrary objects in an application, and they form the basis of the crosscutting available with the CF model. Superimposition of filter modules on an existing object is illustrated in Figure 2.20. Superimposition is responsible to introducing a filtering layer to the object through which messages pass as they go to and from the object. Incoming messages pass in succession through the filters of each of the filter modules superimposed. Note that separate filters are applied to incoming and outgoing messages.

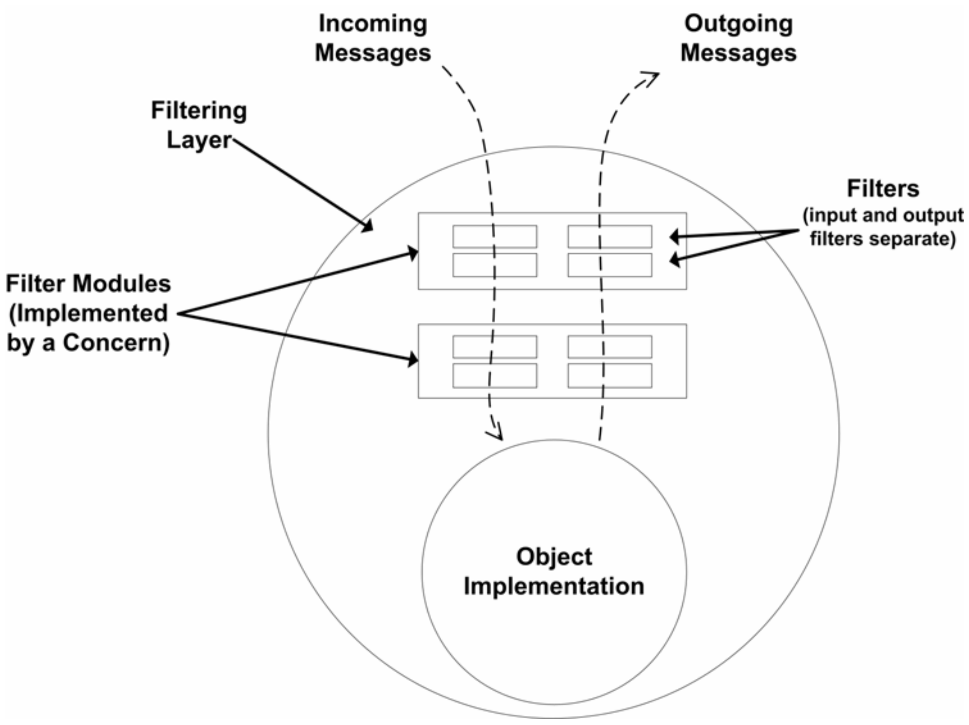


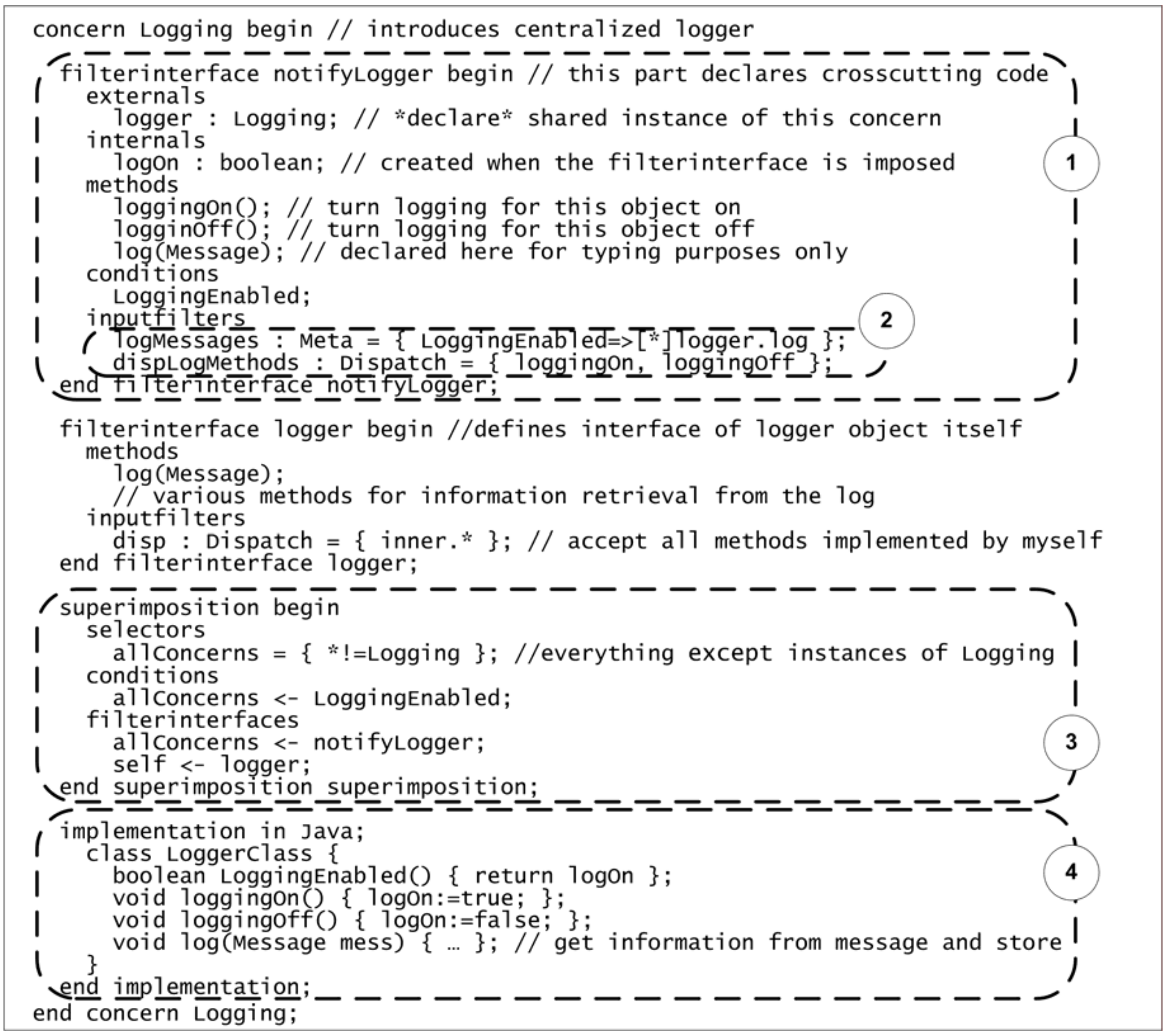
Figure 2.20: Example of superimposition of a filter modules on an object, based on details in [Car'01, Ber'04a].

2.2.5.3 Join Point Model and Means of Identifying Join Points

With Composition Filters, the concern abstraction provides a unit of encapsulation for an aspect, and the join points to which aspect functionality can be applied correspond to the object instances in an application. Object instances are selected in a superimposition by means of *join point selectors*. More recent implementations of the CF model [Car'01] have used the Object Constraint Language (OCL) [OMG'03] and OCL-like languages as the basis for a selection language. OCL is defined as part of the UML standard, and as such provides a reasonably language-independent means of identifying objects. However, work has been applied to overcoming the verboseness of OCL. OCL allows only name-based crosscutting and so has been augmented with regular expression semantics to make crosscuts more succinct.

A sample concern that implements logging crosscutting functionality is shown in Figure 2.21. This example provides a glimpse of how concerns in the CF model are written with ConcernJ [Car'01], which provides support for writing concerns in Java. The point here is not to explain the syntax involved but to highlight the important elements of the concern.

The application of this filter to other objects in the system is governed by the superimposition in the dashed rectangle labelled ‘3’, which applies the filter to all objects in the system except those used to implement the `Logging` concern. This particular example was garnered from the TRESE project website [TRE'04], and precise details of the operation of super-imposition filters is described in [Car'01]. We will refer to this diagram again in the following section.



```
concern Logging begin // introduces centralized logger
  filterinterface notifyLogger begin // this part declares crosscutting code
    externals
      logger : Logging; // *declare* shared instance of this concern
    internals
      logOn : boolean; // created when the filterinterface is imposed
    methods
      loggingOn(); // turn logging for this object on
      logginOff(); // turn logging for this object off
      log(Message); // declared here for typing purposes only
    conditions
      LoggingEnabled;
    inputfilters
      logMessages : Meta = { LoggingEnabled=>[*]logger.log };
      dispLogMethods : Dispatch = { loggingOn, loggingOff };
  end filterinterface notifyLogger;
  filterinterface logger begin //defines interface of logger object itself
    methods
      log(Message);
      // various methods for information retrieval from the log
    inputfilters
      disp : Dispatch = { inner.* }; // accept all methods implemented by myself
  end filterinterface logger;
  superimposition begin
    selectors
      allConcerns = { *!=Logging }; //everything except instances of Logging
    conditions
      allConcerns <- LoggingEnabled;
    filterinterfaces
      allConcerns <- notifyLogger;
      self <- logger;
  end superimposition superimposition;
  implementation in Java;
    class LoggerClass {
      boolean LoggingEnabled() { return logOn };
      void loggingOn() { logOn:=true; };
      void loggingOff() { logOn:=false; };
      void log(Message mess) { … }; // get information from message and store
    }
  end implementation;
end concern Logging;
```

Figure 2.21: Logging concern written in ConcernJ taken from TRESE Composition Filters website [TRE'04].

2.2.5.4 Means of Modifying Join Point Semantics

The means of modifying join point semantics is through the construction of filter modules, which revise the semantics of methods being invoked. Filter modules consist of three elements: filters, supporting methods and supporting objects. The support methods used by a filter can be implemented by a concern directly or accessed from support objects by delegating to the methods of these objects. Method implementations contained in the concern are written using nested classes, whose objects are instantiated by the filter module. The ability to delegate to referenced objects gives filter modules an ability to inherit functionality in a fashion reminiscent of Self [Ung'87]. Filters include a type and decision making logic that determines whether to act on a message. This decision making logic is based in part on the state of the object for which the filter module is intercepting messages, and this state is accessed via helper methods called *conditions*, whose execution returns a Boolean result without influencing the state of the implementing object. Depending on its type, each filter can perform a variety of actions. The simplest filter type is that of *dispatch filters* [Ber'01]. Filters of this type simply accept or reject messages based on Boolean conditions ascertained from condition methods. Rejected messages are passed on to the next filter, or rejected if there is no other filter in the module. Accepted messages get passed to the relevant method for execution. As noted, this method may be implemented by the object whose methods are being intercepted, or the message may be delegated to helper objects referenced by the filter module. The use of delegation is typical when filter modules are applied using superimposition in order to guarantee the availability of a function upon which a filter module is dependent. More complex functionality is available with *wait* filter types [Ber'01] that simplify the creation of message queues and *meta* filter types [Ber'01] that reify messages so that they may be passed to some other method for processing. The availability of different filter types means that not only do composition filters offer strong interception capabilities, but they also provide the abstractions necessary for writing different kinds of interceding functionality in a structured fashion.

In Figure 2.21, the `Logging` concern implements the `notifyLogger` filter module which is identified by a dashed rectangle labelled '1'. In this version of the language, the keyword `FilterInterface` is used to define filter modules. In more recent versions of this tool, the syntax has been made to align with the programming model and so the keyword `FilterInterface` has been changed to `FilterModule` [Ber'04a]. In this filter module, two filters are implemented, and they are identified by the dashed rectangle labelled '2'. The `Meta` filter type reifies incoming messages and passes them to a `log` method implemented by the `class LoggerClass`, identified by the dashed circle labelled '4'. However, this only occurs when the `LoggingEnabled` condition evaluates to true. This condition corresponds to a method also implemented in `LoggerClass`. The second filter, `dispLogMethods`, is of type dispatch. It only intercepts messages exposed by the filter module in the `methods` section. With `dispLogMethods`, messages for `LoggingOn` and

`LoggingOff` are delegated to the `LoggerClass` instance referenced by the filter module. Other messages are passed on to the object being superimposed.

2.2.5.5 Analysis

The CF model provides a much more advanced version of contextual composition. The filter modules of a CF concern provide a programming model for tailorability lacking from contextual composition. Preplanning requirements are eliminated by the transparent application of concerns through superimposition, although the use of the CF model restricts direct access to data members. The programming model can be ported to a variety of languages as the CF semantics are independent of the language implementing; however, interoperability of filter modules and objects written in different languages is not addressed. Also, the filter modules can only be used to augment the behaviour of objects, and it is unclear as to whether concerns can be implemented in non-object oriented languages or not. Reusability involves modifying the aspect, as superimpositions are not specified separately from the filter modules. The focus on OCL does not appear to allow attribute-based property selection as OCL uses object type names. This is a minor issue as the OCL-based mechanism could be extended to use attributes. However, like the PA mechanism, there is no model for separately expressing aspect bindings and aspect crosscutting specifications with attributes.

2.2.6 Summary

A limited set of aspect-oriented mechanisms provide programming models to solve the tailorability problem with contextual composition and avoid the need for preplanning on the part of components to which the aspects are applied; however, as a replacement for contextual composition the PA mechanism appears easier to conceptualise, offers fine-grain crosscutting and suggests better performance. Of the AOP mechanisms investigated in this section, only class composition, pointcut-advice and composition filters were capable of tailoring component behaviour. Crosscutting with object-graph traversal and open class composition did not offer a means of modifying existing behaviour. The programming model presented for class composition introduces the need to divide an application into arbitrary concerns, which are then subject to composition. This is not much use when concerns already align to component boundaries as is the case with contextual composition. Both the PA and CF mechanisms are suitable for addressing the tailorability problem as both make additive changes to behaviour without any preplanning requirements. The need for the CF mechanism to inspect all incoming messages, even those for which no action is taken, suggests that PA mechanisms are lighter weight. Also, the CF model can only affect object behaviour with respect to external messages, where as a PA mechanism does not treat an object as a black-box. Another consideration is that PA mechanisms appear easier to conceptualise. Unlike the CF mechanism, PA mechanisms do not overlap with existing OO mechanisms. For instance, in literature the CF mechanism is described as being able to implement inheritance, but it is unclear whether CF should then replace or complement existing inheritance mechanisms. Such issues do not arise in PA, as the mechanism is distinct from existing OO mechanisms.

In terms of reusability, reuse of aspects continues to involve editing aspects, and in terms of language-independence, aspect-oriented mechanisms fail to address interoperability of aspects and components written in a variety of languages in their programming models. Aspect-oriented mechanisms advocate reusability in accordance with obliviousness in that the aspect, and not the target component, is modified when an aspect is reused. The programming models of aspect-oriented mechanisms leave unaddressed the adoption of attribute-based annotations of components as a means of binding aspects without the need to revise aspect crosscutting semantics. Although the programming models for aspect-oriented mechanisms such as the CF and PA mechanisms make no assumption about the underlying language, they do not address the situation in which an aspect written in one language will be applied to a component written in an unknown language.

2.3 Constraints Component-Oriented Applications Place on Weaving

In examining weaving in the context of software components, we are interested in determining the major restrictions on weaver architecture and in finding an approach that suits the PA mechanism identified in the previous section as a likely replacement to current contextual composition mechanisms. The most immediate candidate for this role appears to be load-time weaving due to its support for clear-box weaving.

The composition of aspects and components should be differed until deployment, as weaving before hand threatens the independent deployment characteristics of components and combines otherwise independent roles in application development such as that of context property writer and component writer. In combining aspects and components, the weaver tangles the implementation of the component with bindings to aspect behaviour. As noted in Chapter 1, tangling forces an aspect and the components to which it is bound to be deployed as a set rather than independently from each other. This problem has been noted elsewhere: "To preserve the property that a component is a unit of deployment, weaving has to take place either within a component before deployment or across component boundaries after deployment." [Szy'02] From the point of view of component development, premature weaving undermines the separation of competence [Pic'03] in which application development is broken up into different roles. Defining development in terms of roles is handy, as each role can be handled by a different party [DeM'03]. For example, the EJB specification [DeM'03] separates component development from context property development and from deployment of context properties in components. Premature weaving prevents these roles from being taken on by different parties. Depending on the weaving mechanism used, the component developer may end up being the context property developer as is the case with compile-time weaving.

A separate issue in composing aspects and components is what granularity of composition is required of weaving, and this influences the mechanism used to bind components to aspect behaviour. Granularity determines whether black-box or clear-box weaving is required. The clear-box / black-box distinction was originally defined with respect to the source code of a program [Fil'00]. A *black-box* technique manipulates components in terms of their public interfaces, while a *clear-box* technique manipulates the parsed language structures used to write these interfaces. Clear-box techniques often offer a richer set of join points, because they provide a better representation of all the structures of a programming language used to write the component. For instance, the difference in granularity between the CF and PA mechanisms is accounted for by support in the latter for clear-box modification of object join points. Typically, components do not provide access to the programming language with which a component was implemented. However, language constructs that are expressed directly in byte code, such as accesses to type members, can be modified [Fil'00]. So with clear-box techniques, instrumentation of byte code is likely, where as black-box techniques can use a proxy technique in which incoming and outgoing messages are intercepted and aspect behaviour applied.

Despite the restrictions that it places on component format, load-time transformation is the most general technology for weaving after deployment due to its variable granularity. Load-time transformation introduced in research on binary component adaptation (BCA)

[Kel'98]. This research demonstrated that load-time transformation was suitable for modifying existing binaries, and binary-level weaving is consistent with the binary format used to distribute components. Load-time transformation was later characterised by work on JOIE, which concluded that it was “a powerful technique in which user-specified transformers add, remove, or change fundamental details of transportable code as it is imported into the local Java Virtual Machine” [Coh'98] The suitability of load-time transformations specifically for aspect-oriented composition was later demonstrated by work on JMangler [Kni'01a]. Work on load-time transformation pointed out two component characteristics that facilitate load-time transformations [Coh'98]. First, component code should be packaged with symbolic information that describes its structure. This facilitates identifying component elements affected by a transformation, and the requirement is met when components contain self-descriptive metadata. Second, the component code should be represented in a format that is easily modified with additions. Here, byte code instructions are advocated, as insertion of new instructions does not adversely affect method state. Byte code has the property that it manipulates data using a stack, and so new operations do not require that the data of existing code be properly stored and retrieved. The major benefit of this emphasis on the use of byte code to express component behaviour is that load-time weaving allows clear-box weaving.

The actual means by which load-time transformation techniques manipulate components as they are loaded varies in terms of transparency and portability. In a survey of load-time transformation techniques, the developers of JMangler [Kni'01b] identified three general means for hooking into class loader architecture pictured in Figure 2.22. The different approaches vary according to the point at which a transformation hook is introduced in the class loader. Original work on component adaptation in BCA exploited hooks introduced into the JVM. Such an architecture would then be coupled to a particular implementation of a component platform. Platform independent tools such as JOIE and the Javassist structural reflection tool [Chi'00] choose to provide a custom class loader implementation. While this mechanism is not coupled to a particular platform implementation, weaving is no longer transparent. Another alternative offered in this survey was to modify the framework APIs that implemented the application class loader. This technique is offered by the JMangler tool, and it avoids the need to modify the platform JVM and simultaneously retains the transparency of weaving as far as the application implementer is concerned, as the application need not use a non-standard class loader.

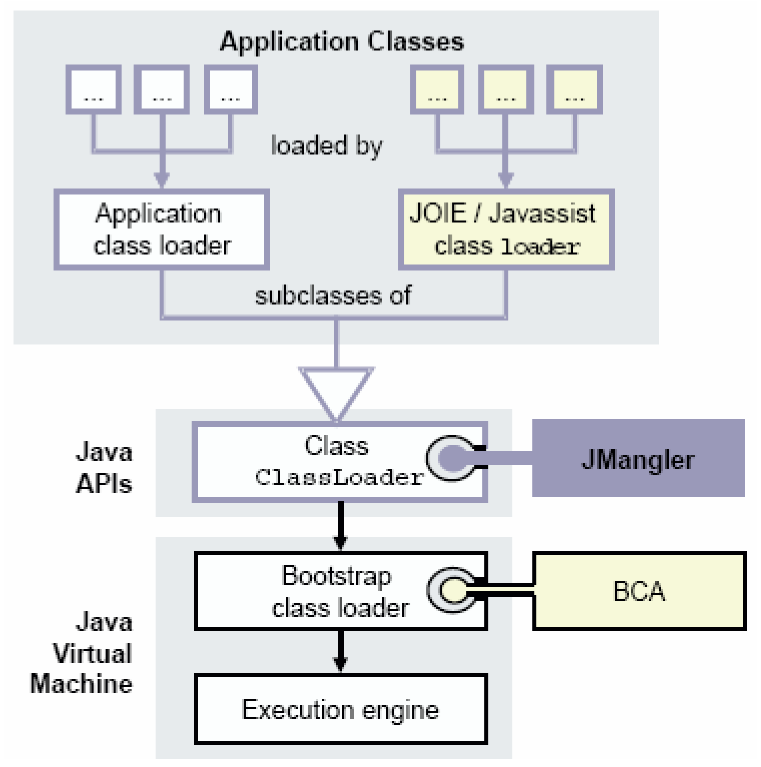


Figure 2.22: "Three ways of hooking into Java's Class Loader Architecture", a graphical representation of load-time transformation implementation options taken from [Kni'01b].

2.4 Summary

In this chapter, we started with the taxonomy of AOP terms and concepts. First, we characterised aspects as a unit of encapsulation that supported the modularization of crosscutting concerns. The aspectual paradox was introduced to describe confusion over how crosscutting functionality could be encapsulated, and this paradox was resolved by pointing out that crosscutting is with respect to the dominant decomposition paradigm within a programming language. The aspect model was presented as a general means for characterising aspect-oriented mechanisms, and this model consisted of the mechanism's join point model, its means of modifying join points and its means of modifying join point semantics. Name-based and property-based crosscutting were introduced to characterise the coupling between crosscutting identifying join points and the implementation of these join points.

Next, we described the following five canonical aspect-oriented mechanisms in terms of their aspect model.

- Pointcut-advice (PA)
- Class composition

- Object-graph traversal
- Open class composition
- Composition Filters (CF)

These mechanisms were then analysed in terms of their ability to replace contextual composition. Only the CF and PA mechanisms could produce suitable changes to component behaviour, and of these mechanisms the PA mechanism was easier to conceptualise, offered finer grained crosscutting and promised better performance. However, while the PA (and CF) mechanisms explicitly made no assumptions about the underlying language of their programming models, the programming models provided did not address interoperability of aspects and components written in a variety of languages. Moreover, none of the mechanisms offered a programming model for separately expressing aspect bindings and aspect crosscutting specifications with attributes.

Finally, we identified load-time weaving as suitable for composing aspects and components after deployment and as offering the clear-box crosscutting required to support a PA mechanism.

Chapter 3 Programming in terms of Aspect-Based Properties

"The secret of a good memory is attention, and attention to a subject depends upon our interest in it. We rarely forget that which has made a deep impression on our minds"
–Tyron Edwards

In this chapter, we present a programming model in which aspect-based properties are used to resolve the reusability and language dependency issues that arise with aspect-oriented mechanisms when they are used to solve the preplanning and tailorability problems with contextual composition. Based on our analysis of aspect-oriented mechanisms in Chapter 2, an AspectJ-like pointcut-advice (PA) mechanism has been chosen as the crosscutting mechanism for aspect-based properties. The problem of reusability is dealt with by splitting the implementation of an aspect-based property from the specification of its binding to a component. To make this split, the programming model for aspect-based properties exploit attribute-based crosscutting in which pointcuts are written with property-based crosscutting in terms of attributes. To complement these crosscuts, the aspect-based property writer provides attribute types that are used to annotate component code by the component writer. Thus, attribute types provide the API for using aspect-based properties, and using attribute types avoids the need to modify the aspect-based property in order to reuse it. Interoperability of aspects and components is guaranteed by use of language-independent AOP. Language-independent AOP is achieved with the PA mechanism of aspect-based properties by referring to join points in terms of their description in component metadata and not the source code used to implement join points. In the context of component metadata, attributes appear as extensions to the metadata description of structures that they annotate. Thus, while the syntax used to annotate types and type members may vary from language to language, the syntax for identifying join points does not. Finally, the programming model for aspect-based properties makes allowances for legacy components by providing custom crosscutting, which allows aspect-based properties to be applied to components not annotated using attribute types.

The programming model used to write aspect-based properties is explained from two views. In section 3.1, we build an overall view of application development involving aspect-based properties. This view is presented in terms of the developer roles and the products passed between these roles. In section 3.2, we concentrate on the aspect model available for writing the crosscutting semantics of aspect-based properties. The aspect model used is similar to the AspectJ PA mechanism discussed in Chapter 2. However, PA specifications are written in XML and separate compilation of aspects and components brings about some limitations to advice.

3.1 Roles and Products

Following the example of the EJB specification [DeM'03], we begin by defining the roles involved in developing software in terms of aspect-based properties. As with EJB roles, each role involved in developing software with aspect-based properties can be carried out by a different party. Interdependencies of these roles are expressed in terms of the products that a role generates and the products that a role exploits but that are generated by other roles. The roles involved in the creation and use of aspect-based properties are as follows:

- Aspect-based property writer
- Component writer
- Application integrator
- Application deployer

The dependencies between each role are expressed in Figure 3.1 as directed arcs. These arcs correspond to one or more products that are generated by the role at which an arrow originates and required by the role to which the arrow points. Details of the products are provided in the following sections in which we list the responsibilities of each role.

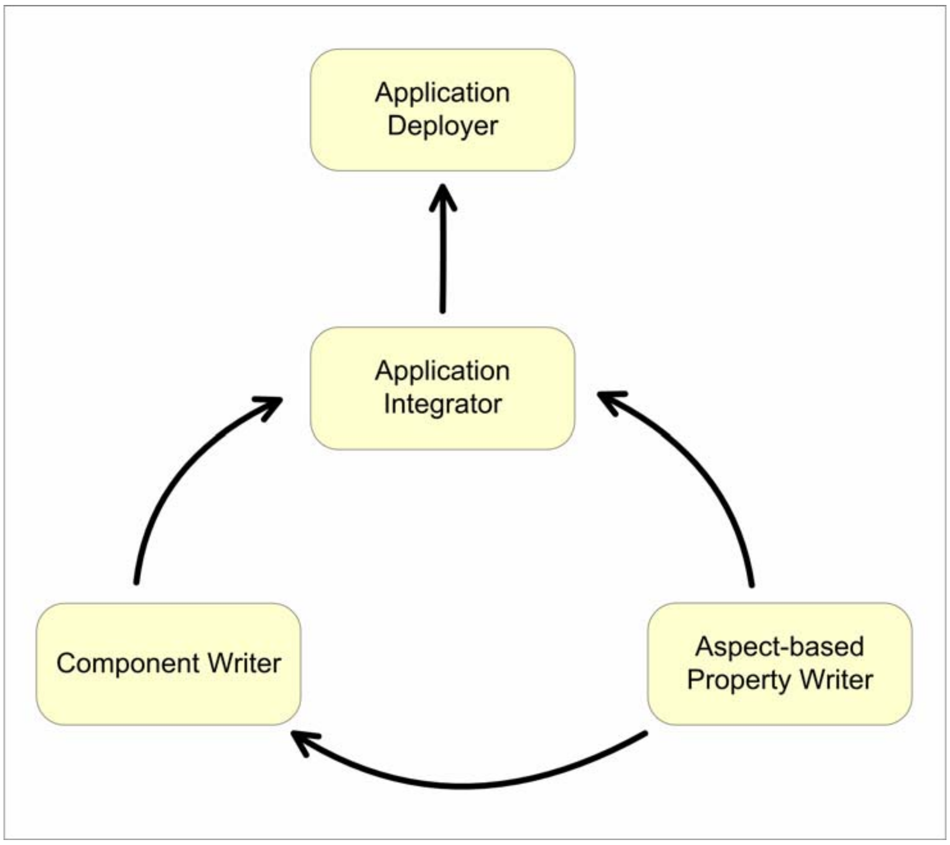


Figure 3.1: Development roles in which aspect-based properties are written and exploited for application development.

3.1.1 Aspect-based Property Writer

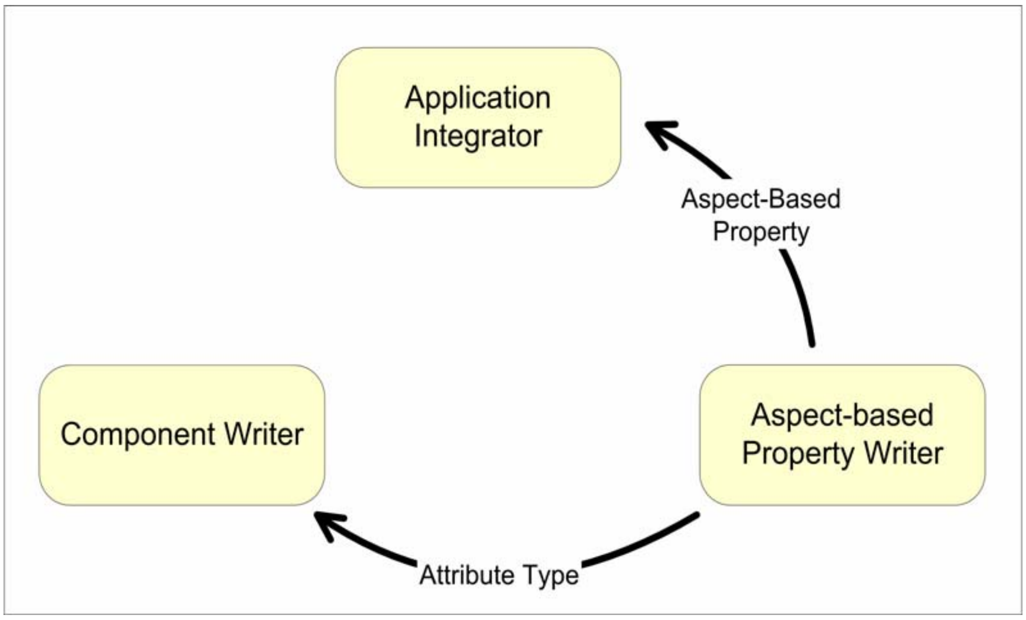


Figure 3.2: Products produced by the aspect-based property writer role and their consumers.

The aspect-based property writer role is responsible for implementing an aspect-based property and attribute types by which the aspect-based property's functionality is accessed.

These products and their consumers are depicted in Figure 3.2. An aspect-based property contains crosscutting functionality that is incorporated into a complete application. Aspect-based properties are made available to the application integrator for distribution with the final application. Attribute types provide an API by which aspect-based functionality is accessed. Attribute types implement attributes that the component writer uses to annotate components in order to specify component to aspect-based property bindings.

In the following subsections, we elaborate on the architecture of the aspect-based property and attribute type products.

3.1.1.1 Architecture of Aspect-Based Properties

Aspect-based properties are aspects defined using a pointcut-advice mechanism. The choice of a pointcut-advice mechanism reflects the results of our analysis of aspect-oriented mechanisms presented in Chapter 2. Recall that in section 2.2.6 we noted two potential replacements for contextual composition, and these were the CF and PA mechanisms. Of these mechanisms, the PA mechanism offers semantics that are easier to conceptualise, provides finer grained crosscutting, and promises better performance. In terms of granularity, PA semantics support clear-box crosscutting allowing, for example, both field access and method calls to be manipulated. Also, there is a clear-cut distinction between a PA mechanism and existing object-oriented mechanisms. This distinction is not as clear with composition filters. For instance, the CF programming model describes composition filters as able to simulate inheritance [Ber'04a], which blurs the distinction with inheritance. In terms of performance, the CF mechanism intercepts each message coming into an object, whether or not the message triggers filter behaviour. A PA-based alternative can avoid this indirection by adding new behaviour to only those methods that are affected by a crosscutting concern.

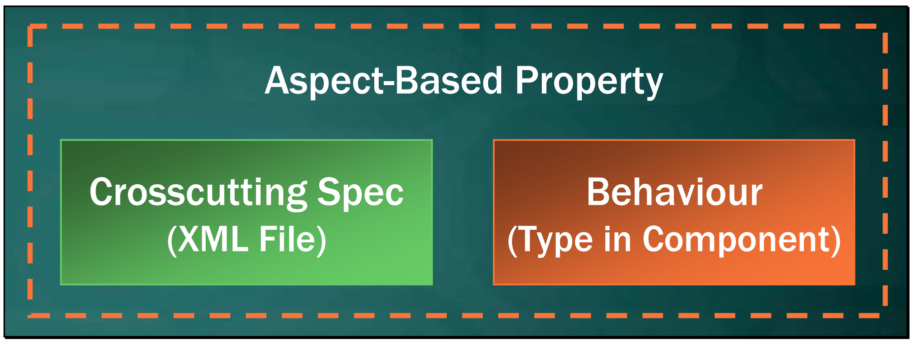


Figure 3.3: Architecture of an aspect-based property.

Architecturally, an aspect-based property consists of a type implementation and an attribute-based crosscutting specification that is written in XML-based. Figure 3.3 provides an overview of this architecture. The type implementing aspect behaviour is distributed as a component in a format consistent with the component platform with which aspect-based properties are being used. While the architecture of aspect-based properties is not component platform specific, the examples in this chapter refer to an implementation of aspect-based properties for Microsoft's CLI [Ecm'03b], which reflects our selection of the CLI as the target platform for the prototype implementation of aspect-based properties described in Chapter 4. Crosscutting specifications that consist of pointcut and advice declarations are distributed in a separate file with an .XML extension. Attribute-based crosscutting means pointcuts identifies join points in term of attributes. Advice statements cross reference the pointcuts with methods that implement advice behaviour.

The decision to use XML reflects our interest in accommodating existing programming languages and software component architectures, while allowing for future changes in language syntax. The purpose of storing crosscutting specifications in XML format is to avoid the need to modify the binary format of components to store these crosscutting semantics. Extensible metadata is another alternative for storing PA specifications. Exploiting extensible metadata requires some sort of language-level support capable of interpreting the PA specifications and storing them in metadata in a standard format. For instance language extensions might be used to express PA specifications, and instead of influencing compilation these extensions might simply record PA specifications in extensible metadata. Unfortunately, such restrictions of aspect-based property programming would interfere with the language-independence that we seek, as only programming languages with the appropriate extensions could be used to write aspect-based properties. Rather than adding extensions, PA specifications could be expressed in the form of attributes, which are emitted directly to extensible metadata [Ecm'03b]; however, the literature makes no comment on the use of attributes in this manner. Our XML format for PA specifications anticipates the future appearance of these systems as well as future languages that provide first class support for a PA mechanism, because XML provides a standard input format for weavers to which attribute-based PA specifications and those generated by language extensions can be mapped.

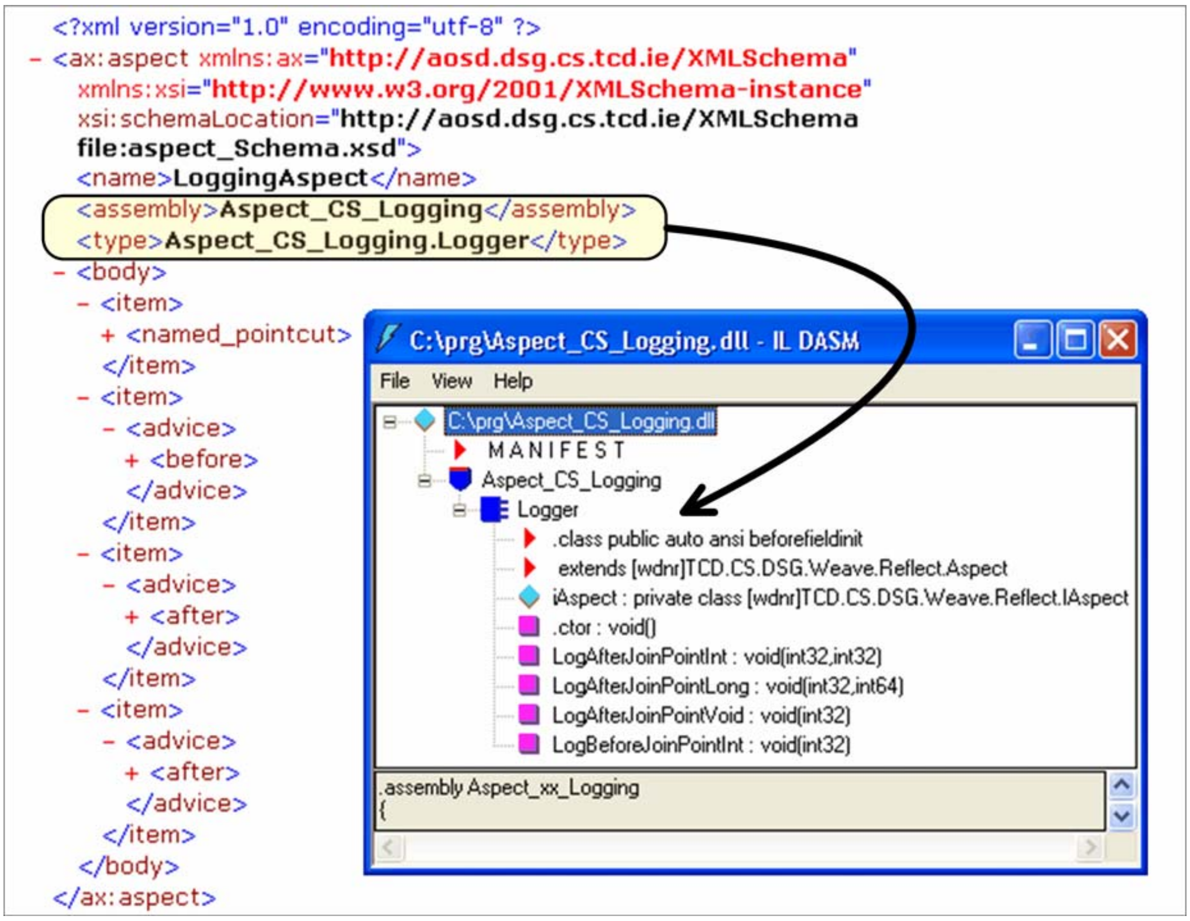
The PA specifications available with aspect-based properties are based on those of AspectJ V1.0.6 [Asp'02]. AspectJ was adopted, because AspectJ language semantics are published in a programming guide. This particular version was chosen, because prior to V1.0 the language semantics were unstable. For example, the keywords of the version 0.8 release [Kic'01b] use a different syntax to previous versions as well as those that followed. Even earlier releases [Lop'97] use different language semantics. Variations in the join point model during language evolution have been noted elsewhere [Kic'01b]. Version 1.0 represents the first stable specification, as subsequent evolution has made only additive changes to the aspect model used by the language. For instance, version 1.1 [Lad'03] makes available more primitive pointcut designators. The language specification in the AspectJ V1.0.6 programming guide [Asp'02] is sufficiently detailed to allow its conversion to XML in a reasonably methodical manner [Laf'02b]. That is not to say that the language semantics documentation is perfect. First, there are some inconsistencies. The specification for the wildcard character used for property-based crosscutting contradicts itself[2]. Second, crosscutting in terms of attributes was not supported. The XML for aspect-based properties includes some minor extensions that allow attributes to be used, which we describe in 3.2.2 when we discuss the use of XML to select join points.

A sample aspect-based property is shown in Figure 3.4. This aspect-based property provides logging to join points identified in the XML specification on the left-hand side of the diagram. In this example, the XML cross-references a type in a CLI assembly providing the aspect behaviour, and a graphical view of that assembly appears on the right-hand side of the figure. This view is generated by Ildasm [Mic'04a], which provides details on the internal structure of the CLI component including the types that the assembly holds as well as the type's members. The aspect-based property identifies its behaviour according to both the implementing component and the implementing type. These details appear at the beginning of the XML and are highlighted in the figure. The XML format is governed by a schema [Laf'02b] written in W3C XML Schema Language [Fal'01]. Although the XML in Figure 3.4 uses a local copy of the schema, it is also available on the internet[3]. We discuss this schema in detail in section 3.2, provide details of its derivation

[2] In the context of a type pattern [Asp'02]: "There is a special type name, `*`, which is also a type pattern. `*` picks out all types, including primitive types. So `call(void foo(*))` picks out all call join points to void methods named `foo`, taking one argument of any type." But in the next paragraph "The `*` wildcard matches zero or more characters". In this case , `call(void foo(*))` picks out all call join points to void methods named `foo`, taking one **or zero** argument**s** of any type."

[3] http://aosd.dsg.cs.tcd.ie/XMLSchema/aspect_Schema.xsd

from the AspectJ V1.0.6 semantics in Chapter 4, and present the actual schema in Appendix A.

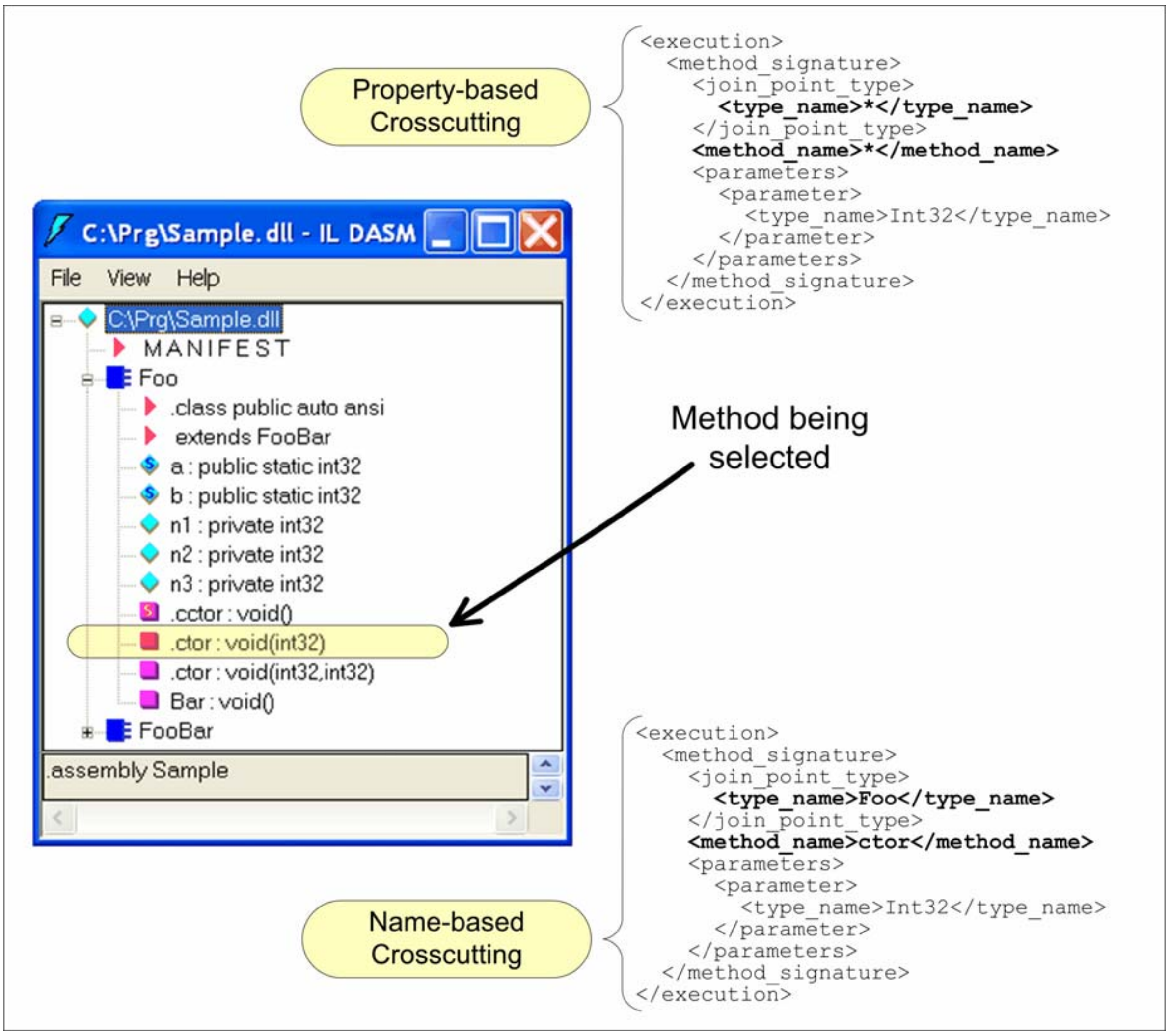


```
  <?xml version="1.0" encoding="utf-8" ?>
- <ax:aspect xmlns:ax="http://aosd.dsg.cs.tcd.ie/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://aosd.dsg.cs.tcd.ie/XMLSchema
    file:aspect_Schema.xsd">
    <name>LoggingAspect</name>
    <assembly>Aspect_CS_Logging</assembly>
    <type>Aspect_CS_Logging.Logger</type>
  - <body>
    - <item>
      + <named_pointcut>
      </item>
    - <item>
      - <advice>
        + <before>
        </advice>
      </item>
    - <item>
      - <advice>
        + <after>
        </advice>
      </item>
    - <item>
      - <advice>
        + <after>
        </advice>
      </item>
    </body>
  </ax:aspect>
```

```
C:\prg\Aspect_CS_Logging.dll - IL DASM
File  View  Help
C:\prg\Aspect_CS_Logging.dll
    MANIFEST
    Aspect_CS_Logging
        Logger
            .class public auto ansi beforefieldinit
             extends [wdnr]TCD.CS.DSG.Weave.Reflect.Aspect
            iAspect : private class [wdnr]TCD.CS.DSG.Weave.Reflect.IAspect
            .ctor : void()
            LogAfterJoinPointInt : void(int32,int32)
            LogAfterJoinPointLong : void(int32,int64)
            LogAfterJoinPointVoid : void(int32)
            LogBeforeJoinPointInt : void(int32)
.assembly Aspect_xx_Logging
{
```

Figure 3.4: Sample aspect-based property crosscutting specifications and behaviour implementation.

Although aspect-based properties exploit the same primitive AspectJ pointcut designators that were presented in Chapter 2, attribute-based crosscutting is used to make aspect-based properties reusable without modification. Recall from the description of crosscutting with a PA mechanism in section 2.2.1.3 that property-based crosscutting consistent with obliviousness involved identifying implementation details common among join points being selected. These details were then used as the argument for the pointcut that selects the join points. This was in contrast to name-based crosscutting in which join points are identified individually based on the metadata details of the join point's implementation. With execution and call join points, these details correspond to the signature of the method called or executed as well as the type containing the implementation. For field accesses, the field type and a containing type are important. Name-based and property-based crosscutting with strong obliviousness characteristics are available with the aspect-based property programming model; however, their use is limited to customization steps used under exceptional circumstances by the application integrator, which we discuss in section 3.1.4. When writing aspect-based properties, the correct approach to writing pointcuts is to use attribute-based crosscutting. This is a subset of property-based crosscutting in which attributes are used in place of method signatures, field signatures and type specifications that are normally used to parameterise primitive pointcut designators. The use of attributes is emphasised in order to decouple the pointcut specification from the implementation details of the component to which it is applied. Splitting binding from aspect implementation has been used before in order to improve the reusability of a particular aspect implementation [Pic'03]. As we will see, the final binding is left to the component writer.

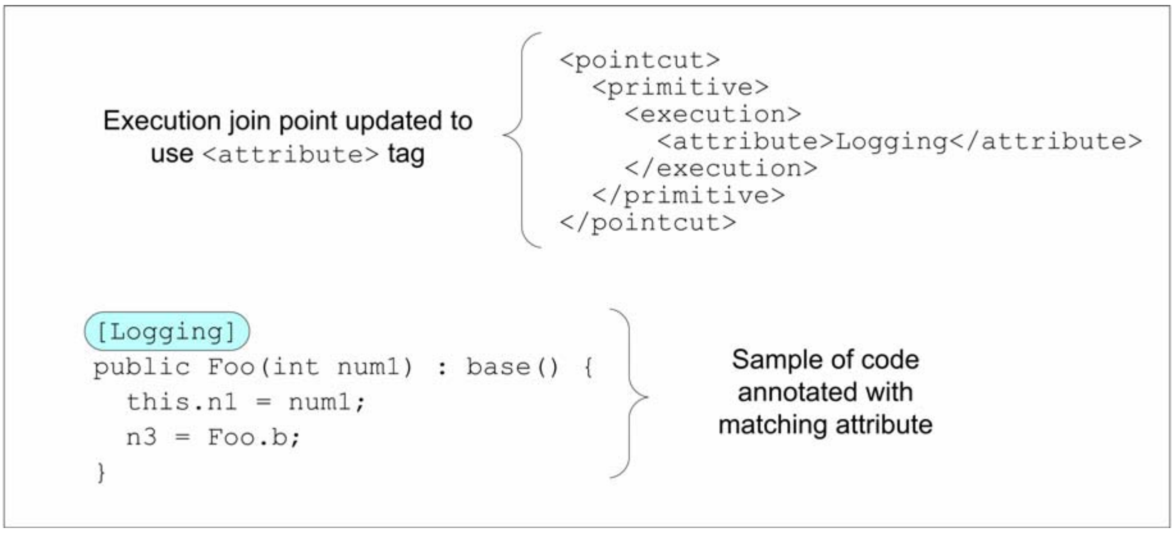


```
<execution>
  <method_signature>
    <join_point_type>
      <type_name>*</type_name>
    </join_point_type>
    <method_name>*</method_name>
    <parameters>
      <parameter>
        <type_name>Int32</type_name>
      </parameter>
    </parameters>
  </method_signature>
</execution>
```

```
<execution>
  <method_signature>
    <join_point_type>
      <type_name>Foo</type_name>
    </join_point_type>
    <method_name>ctor</method_name>
    <parameters>
      <parameter>
        <type_name>Int32</type_name>
      </parameter>
    </parameters>
  </method_signature>
</execution>
```

Figure 3.5: Examples of strong obliviousness with name-based and property-based crosscutting.

To contrast the crosscutting with strong obliviousness and attribute-based crosscutting, we present samples of each approach in Figure 3.5 and Figure 3.6. In Figure 3.5, implementation details in the form of a method signature are used to identify execution join points involving one of the constructor methods in class `Foo`. These details appear in the XML on the right-hand side of the figure. At bottom of the figure, name-based crosscutting identifies the class defining the constructor, its method name, and the parameter type. The type names and the convention of using `ctor` as the name of constructor methods are specific to our CLI prototype, but the form of the method signature, i.e., the XML tag structure, is consistent across implementations of aspect-based properties. In contrast, the property-based crosscut picks the presence of a single `Int32` parameter as the distinguishing characteristic of methods whose execution will be selected. The use of either approach depicted in Figure 3.5 is considered bad practice when specifying the crosscutting semantics of aspect-based properties. As the attribute-based crosscutting embraced by aspect-based properties uses a style in which signatures and type specifications are expressed with attributes. Such an example is shown in Figure 3.6, where execution join points are selected based on the method executing at the join point being annotated with an attribute. The pointcut specification, again written in XML, matches join points that correspond to methods annotated with an attribute with the name `Logging`. The figure provides sample component source in which the `Logging` attribute type is applied to the `Foo` constructor that was highlighted in Figure 3.5. Such labelling is the responsibility of the component writer, and this role discussed in section 3.1.2.

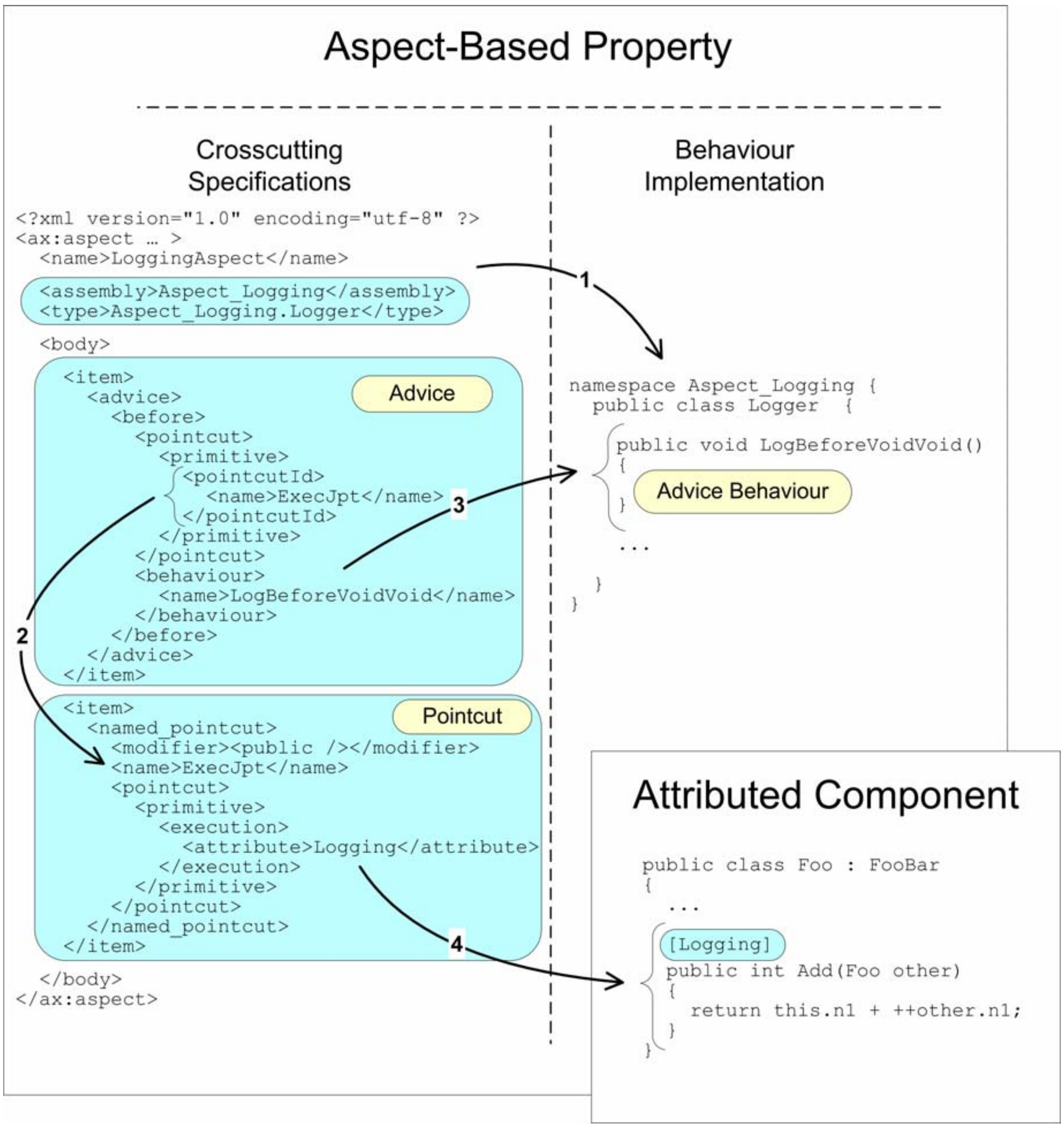

Execution join point updated to use <attribute> tag
Sample of code annotated with matching attribute

```
<pointcut>
  <primitive>
    <execution>
      <attribute>Logging</attribute>
    </execution>
  </primitive>
</pointcut>
```

```
[Logging]
public Foo(int num1) : base() {
  this.n1 = num1;
  n3 = Foo.b;
}
```

Figure 3.6: Attribute-based crosscutting equivalent Figure 3.5 and required constructor annotation.

It is important to point out that while the attribute name is referenced in the crosscutting specification of an aspect-based property, this attribute type's not required by the implementation of the aspect-based property. Only the attribute's name is referenced in the XML-based crosscutting specification. The XML is not compiled or linked, and so the aspect-based property does not require the attribute name being referenced to actually be defined. Matching between attributes specified in XML and the attribute applied to a component is by name, so the name of the attribute used to annotate component source must be consistent with the name used in the crosscutting specification of an aspect-based property. Consistency is maintained by having the aspect-based property writer provide an implementation of the attribute with the correct name for use by the component writer. This implementation is called an attribute type. We discuss attribute types in the next section.



```
<?xml version="1.0" encoding="utf-8" ?>
<ax:aspect … >
  <name>LoggingAspect</name>
  <assembly>Aspect_Logging</assembly>
  <type>Aspect_Logging.Logger</type>
  <body>
    <item>
      <advice>
        <before>
          <pointcut>
            <primitive>
              <pointcutId>
                <name>ExecJpt</name>
              </pointcutId>
            </primitive>
          </pointcut>
          <behaviour>
            <name>LogBeforeVoidVoid</name>
          </behaviour>
        </before>
      </advice>
    </item>
    <item>
      <named_pointcut>
        <modifier><public /></modifier>
        <name>ExecJpt</name>
        <pointcut>
          <primitive>
            <execution>
              <attribute>Logging</attribute>
            </execution>
          </primitive>
        </pointcut>
      </named_pointcut>
    </item>
  </body>
</ax:aspect>
```

```
namespace Aspect_Logging {
  public class Logger  {
    public void LogBeforeVoidVoid()
    {
    }
    ...
  }
}
```

```
public class Foo : FooBar
{
  ...
  [Logging]
  public int Add(Foo other)
  {
    return this.n1 + ++other.n1;
  }
}
```

Figure 3.7: Overview of a crosscutting specification in the context of an aspect-based property.

To provide additional clarity as to what is and what is not an aspect-based property, we refer to Figure 3.7 in which the implementation and crosscutting specification of an aspect-based property are pictured along with an attributed component. The attributed component makes use of aspect-based property functionality via an attribute type. Attributed components will be discussed in more detail in section 3.1.2 and so we concentrate on the aspect-based property here. Aspect-based properties are not bound to a particular attributed component, and so the attributed component given is only representative. This particular aspect-based property defines logging. The implementing type is referenced by the crosscutting specification according to the type's containing component and the type's name. This reference is indicated with arrow 1. Methods in the implementing type are referenced on an advice by advice basis. For instance, arrow 3 points out a method that is used to implement a before advice statement. Note that though aspect-based property behaviour in Figure 3.7 is written in C#, any language can be used to express the aspect-based property behaviour. Also note that the advice references implementing methods explicitly by name and implicitly by method parameters. The required parameters for an advice statement can be determined by examining what typed formal parameters are bound by the advice statement and the named pointcut that advice references. A reference from advice to a pointcut is shown by arrow 2. For simplicity, this example does not include a typed formal parameter. Typed formal parameters are passed to method implementing advice, and they will be presented in section 3.2. Finally, arrow 4, points out a method whose execution generates join points that match the pointcut of the aspect-based property. Although the component writer is responsible for annotating the component with attributes recognised by the aspect-based property, aspect-based property used is not know until the application integrator chooses a particular attributed component and aspect-based property for an application. We will discuss the application integrator role in section 3.1.3.

3.1.1.2 Attribute Types

An attribute type is an API for accessing aspect-based property functionality. The attribute type allows a component writer to bind a component to an aspect-based property without the need for the component writer to see the aspect-based property's implementation or modify the aspect-based property's crosscutting specifications. To do this, the attribute type provides an implementation of an attribute referenced by the crosscutting specifications of aspect-based properties that is used to by the component writer to annotate component source.

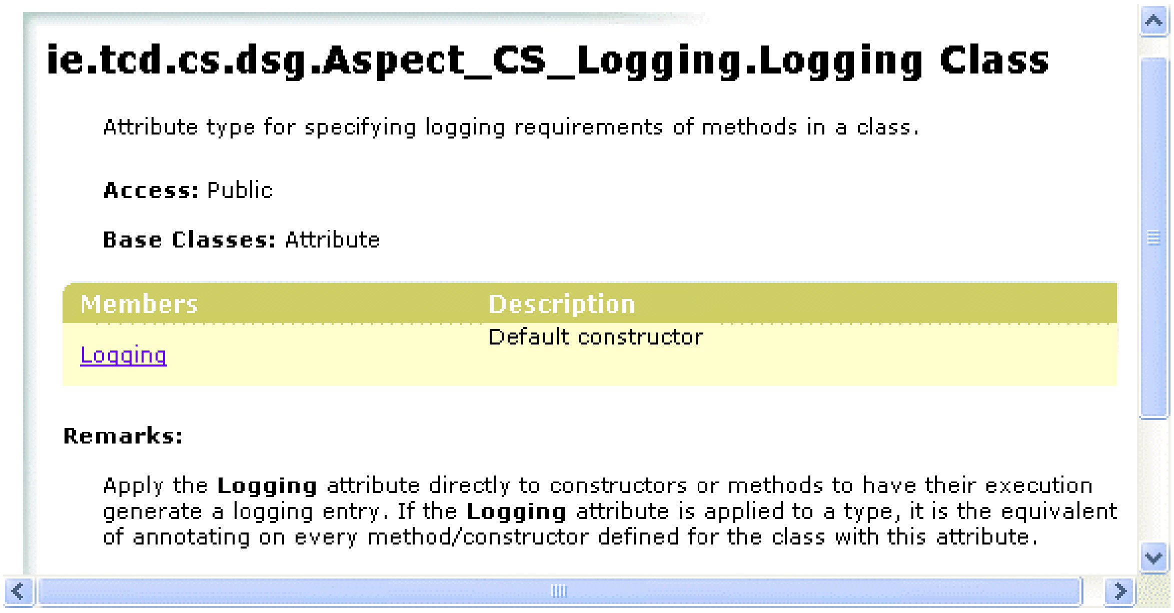
Physically, all attribute types referenced by an aspect-based property are distributed in a single component. Details of how an attribute type is used to annotate a component, as well as the functionality accessed by that attribute type, are provided in the attribute type's documentation. Such documentation may take the form of a class overview that characterises where the attribute may be placed and the semantics of this placement. In practice, the attribute type's implementation and its written semantics can be merged by placing the semantics in source code comments. These comments can then be separated at compilation using a standard API generator such as Javadoc [Sun'04] or Doxygen [van'04], depending on the platform for which aspect-based properties are implemented. In addition to having written documentation, attribute types may be tailored in so far as their implementation may allow them to be restricted from annotating certain types of structural elements, i.e., an aspect-based property writer may tailor the implementation of an attribute type to prevent the attribute type from being used in a manner inconsistent with the aspect-based property's crosscutting specification. The restrictions available are platform dependent. We explain these restrictions when we discuss the implementation of an attribute type.

Note that attributes types provided by aspect-based properties cannot be parameterised. This is in contrast to the approach of AspectJ2EE [Coh'04], an aspect-oriented approach used to solve tailorability, though not preplanning, with context-based properties. AspectJ2EE parameterises aspect-component bindings in order to improve reusability. Specifically, abstract pointcuts can be specified in the declarative binding between an aspect and a component. However, we wish to avoid abstract pointcuts in order to simplify the component writer role. The difficulty is that abstract pointcuts require that the user understand concepts beyond the scope of a non-AOP language such as pointcut specification, and this extra learning will impede adoption. A second kind of parameterization used in [Coh'04] was the specification of initial values for an aspect. There is precedent for allowing these initial values to be assigned to fields in objects corresponding to attribute type when the attributes are used to annotate source code [Sch'02], but such parameterization is not used with aspect-based properties. Where initial values are required for proper aspect-based property operation, these can be obtained directly from type data members using clear-box crosscutting. In this case, there should be an attribute type to annotate the data member that provides the initial value.

The implementation of a very simple attribute type to access a logging aspect-based property is shown in Figure 3.8, and its documentation is shown in Figure 3.9. Figure 3.8 shows the implementation of the attribute type, which is written in C# and targeted at the CLI component platform. The language used in this example reflects the choice of platform made for our prototype, and it is not meant to imply that attribute types can only be implemented for the CLI platform. The declaration that appears in bold in Figure 3.8 restricts the program structures that can be annotated by this attribute to class types, methods and constructor methods. This is consistent with the attribute semantics described in the code comments of Figure 3.8. Note the lengthy namespace in which attribute types are defined. To avoid naming conflicts between attribute types implemented by different organisations, the name of an attribute type should have a prefix corresponding to the organisation producing the attribute type. An HTML-based description of the attribute type is rendered in Figure 3.9. This description was created programmatically by extracting and formatting source code comments from the implementation in Figure 3.8. In this case, the semantics indicate that logging is bound to the execution of methods annotated with the attribute defined by this type.

```
using System;

namespace ie.tcd.cs.dsg.Aspect_CS_Logging
{
  /// <summary>
  /// Attribute type for specifying logging requirements of methods
  /// in a class.
  /// </summary>
  /// <remarks>
  /// Apply the <b>Logging</b> attribute directly to constructors or
  /// methods to have their execution generate a logging entry.
  /// If the <b>Logging</b> attribute is applied to a type, it is the
  /// equivalent of annotating on every method/constructor
  /// defined for the class with this attribute
  /// </remarks>
  [AttributeUsage(AttributeTargets.Method
                 |AttributeTargets.Class|
                  AttributeTargets.Constructor)]
  public class Logging : Attribute
  {
    /// <summary>
    /// Default constructor
    /// </summary>
    public Logging(){}
  }
}
```

Figure 3.8: Sample attribute type implementation for the CLI.



ie.tcd.cs.dsg.Aspect_CS_Logging.Logging Class

Attribute type for specifying logging requirements of methods in a class.

Access: Public

Base Classes: Attribute

Members	Description
Logging	Default constructor

Remarks:

Apply the **Logging** attribute directly to constructors or methods to have their execution generate a logging entry. If the **Logging** attribute is applied to a type, it is the equivalent of annotating on every method/constructor defined for the class with this attribute.

Figure 3.9: Sample API specification for an attribute type.

3.1.2 Component Writer

The component writer has specialised knowledge about components being implemented that allows him to properly bind a component to an aspect-based property. These bindings take the form of attributes annotated to the declarations of structural elements of a component. Figure 3.10 provides an overview of the products required and generated by the component writer role. An attribute type provided by the aspect-based property writer role provides an implementation of the attribute used to annotate the component being implemented by the component writer. In section 3.1.1.2, we explained that attribute types are independent of the aspect-based property that implements the functionality attributes reference, and we explained that attribute types include a description of the semantics of the attributes provided. When an attribute type is applied to a component, the result is called an attributed component. An attributed component does not yet contain the crosscutting functionality that corresponds to the attribute type. This binding is carried out by the application deployer role, which is discussed in section 3.1.4.

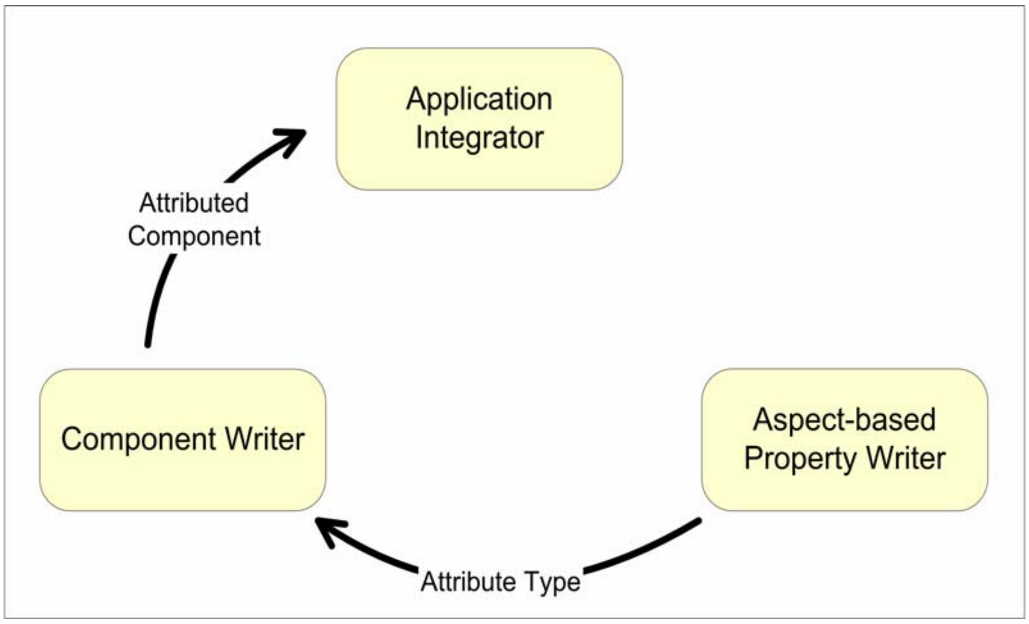
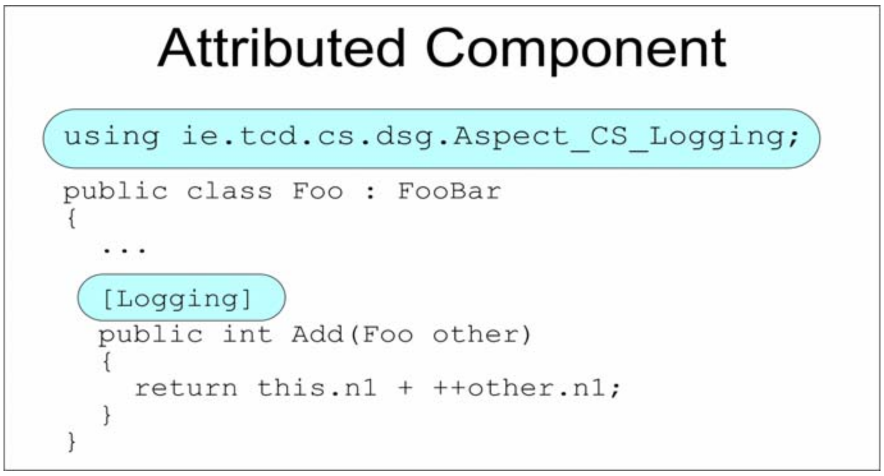


Figure 3.10: Overview of products produced and consumed by the component writer role.

The attribute types used by the component writer role were discussed in section 3.1.1.2, so we focus in this section on the characteristics of an attributed component.

3.1.2.1 Attributed Components

An attributed component corresponds to a component to which attributes have been added using one or more attribute types. These bindings specify when aspect-based properties should intercede during the execution of an application that makes use of the attributed component. However, these bindings do not allow the component writer to select specific aspect-based properties. The component writer is expected to have a good knowledge of the component being annotated, because the aspect-based property writer relies on the component writer to avoid conflicts between the attributed component and the aspect-based property with which the attributed component is woven. For instance, an aspect-based property that supports persistence relies on the component writer to know which specific objects need to be persisted so as not to cripple application performance with unnecessary updates to a persistent data store. The resolution of aspect-aspect conflicts is a separate problem that is referred to as the "composability problem" [Szy'02]. The composability problem has been examined in recent research [Kni'01a], but a means of completely avoiding the problem is not available. In our programming model, the component writer is expected to do his best to spot and address conflicts between attribute types applied to the same component; however, complete elimination of the composability problem should be verified during integration testing of the final application.


Attributed Component

```
using ie.tcd.cs.dsg.Aspect_CS_Logging;
public class Foo : FooBar
{
  ...
  [Logging]
  public int Add(Foo other)
  {
    return this.n1 + ++other.n1;
  }
}
```

Figure 3.11: Attribute component of Figure 3.7 updated to include the attribute type's namespace.

Since attributes are applied using source code annotations, the component writer role only applies to the creation of new components or those for which source code is available. An example application of an attribute type was shown in Figure 3.7. An expanded version of this example is shown above in Figure 3.11. In this version of the attributed component, we use a `using` declaration to avoid having to use the fully qualified name when using the `Logging` attribute. Without the `using` declaration, the attribute used would take the form `[ie.tcd.cs.dsg.Aspect_CS_Logging.Logging]`. This lengthy type name reflects the use of fully qualified type names to distinguish between attribute types that correspond to different crosscutting functionality.

Note that our programming model does not consider versioning of attribute types. The implementations of aspect-based properties are independent to that of an attributed component in that the type implementing aspect-based property behaviour can vary. The crosscutting specification can also vary, so long as the semantics of the attributes published with the attribute type do not change. Changes to the semantics available with an aspect-based property would require a new attribute type. The static nature of attribute types reflects the difficulties for an attributed component were the attribute type to change. Changes to the semantics of an attribute type would in turn require a re-evaluation of the bindings used by a particular attributed component to ensure that they were still correct. Preventing the attribute type from changing avoids the need for such re-evaluations.

A major benefit of using attributes to specify component to aspect-based property bindings is that attributes are consistent with language-independence in which components can be written in any language. This is because attributes can be applied to the source code of any component regardless of implementing language. In justifying the selection of an XML format for crosscutting specifications, we pointed out that attribute annotations are emitted directly to component metadata extensions. Specifically, attributes are stored in a standard format as metadata extensions to the components to which they are bound. Attributes also have broad language support, as component platforms typically stipulate their support at the language level as is the case with the CLI [Ecm'03b] and J2EE [Blo'03]. For example, while the syntax used to place attributes varies between languages that have been implemented for the CLI, such as SML.NET [Ken'03], C# [Mic'01], and VisualBasic.NET [Mic'04d], the same attribute type implementation can be used for attribute declarations regardless of the language being annotated, provided the component platform is the same. Furthermore, component platforms complement extensible metadata with an API that allows the inspection of the metadata extensions applied to a particular component. Thus, the specification and interpretation of attributes applied to a component has no influence on component programming language. The decision to rely on attributes and component metadata for language-independence was chosen as it scales well by avoiding modifications to language tools. These claims to language-independence are verified in the evaluation of Chapter 5.

In our programming model, the attribute type must be included with the attributed component when it is distributed. The attribute type is required to indicate to the application integrator which aspect-based properties are required. Proper operation of an attributed component requires that it be bound to an aspect-based property when the overall application is assembled. The application integrator will likely have to find an aspect-based property based on what is required by the attributed component, since the application integrator cannot be expected to automatically have the required aspect-based property. Moreover, the set of aspect-based properties is not fixed. Recall that the ability to continually create new aspect-based properties is the basis of our solution to the tailorability problem. Determining aspect-based property requirements from the attributed component would involve using the reflective API of a component platform to extract attribute types applied to an attributed component. However, this is far from ideal. For example, with the CLI platform, extraction of such metadata extensions requires the instantiation of the attribute type. In the absence of an attribute type, this operation will fail. Even with a means of inspecting attribute types that avoids instantiation of these types, without additional information it would be difficult to distinguish aspect-based property-related attributes from metadata extensions included for other purposes. For instance, there are examples in which attributes are used to embed test framework information [New'02]. Thus, we assume in our model that attribute types required for a component to link properly are included with that component when it is distributed.

3.1.3 Application Integrator

The application integrator is responsible for assembling an application that consists of attributed components, supporting aspect-based properties and application-specific customizations. The products required and produced by the application integrator are summarised in Figure 3.12. Among the items produced are the aspect-based properties and attributed components used by a particular application. They are not so much produced as they are passed on. The attribute types used by attributed components are included in the items passed on as well, as they are included with the attributed component to guarantee proper application execution. We will discuss the continued purpose of attribute types in the next section when we examine the application deployer role.

Figure 3.12 also notes products created by the application integrator. These are the integration components and custom crosscutting. These products are more ad hoc in nature than attribute types, attributed components and aspect-based properties. In fact, integration components and custom crosscutting are not reusable outside the context of a particular application. Integration components address application specific functionality required to integrate attributed components. Custom crosscutting represents a second, more extreme form of customization in which changes to an aspect-based property’s crosscutting specifications are made. Customization of crosscutting specifications allows the application integrator to deal with legacy components that have not been annotated with attributes, but that must adopt crosscutting functionality to correctly integrate with the application.

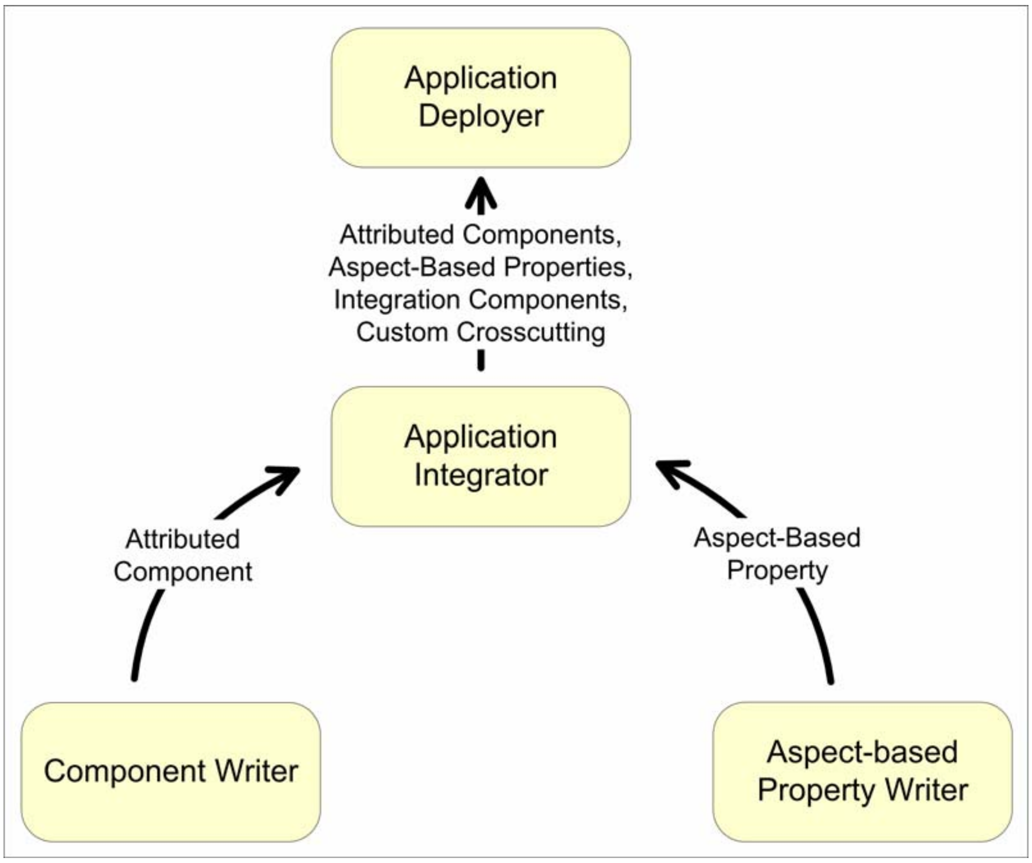

Application Deployer
Attributed Components, Aspect-Based Properties, Integration Components, Custom Crosscutting
Application Integrator
Attributed Component
Aspect-Based Property
Component Writer
Aspect-based Property Writer

Figure 3.12: Overview of products produced and consumed by the application integrator role.

In the remainder of this section we will examine an integration task in which an application is composed with profiling crosscutting functionality. In the first scenario, the application integrator implements the entire system with attributed components and aspect-based properties. This solution allows us to highlight the interoperability of components and aspects that language-independent AOP provides to our programming model. In the second scenario, the application integrator must perform considerable customization to integrate the system. This scenario allows us to discuss how tailorability and preplanning problems are dealt when our programming model is applied to legacy components not developed by the component writer role.

3.1.3.1 Using Attributed Components and Aspect-Based Properties Only

Let us start by introducing a very simple application integration task. A variety of programs include diagnostic backends that allow examination of program state. Take the example of real-time controllers that execute a control program at a fixed interval. An administrator of such controllers may be interested in the amount of time in each interval is spent executing control logic. Such profiling concerns can be viewed as crosscutting concerns. The methods whose execution time is of concern are crosscut by the same timing function. In our scenario, an application integrator is interested in assembling an application with strong self-profiling characteristics. That is, the application should provide dynamic updates of execution times of key systems. The application is the real-time controller discussed above. In this case, the elements being profiled are the execution of controller logic, inter-controller communications, and logic updates.

In the ideal scenario, the application integrator could satisfy application requirements, simply by selecting the appropriate attributed components and aspect-based properties and using them 'as is'. The attributed components chosen should address primary application functionality. In the case of a real-time controller, attributed components should handle controller logic execution, inter-controller communications, the reconfiguration of the controller program, and any other subsystems of the controller. The attributed components should also include bindings to crosscutting functionality that address the profiling crosscutting concern. We would expect each attributed component to include an attribute type corresponding to the profiling used. Preferably, the same attribute type would be used by each attributed component. This would allow for a consistent implementation of the mechanism used to report profiling information to an administrator. We would expect the aspect-based property to place the profiling information in a central location for inspection, but the actual reporting mechanism may vary considerably with some aspect-based properties simply updating a file or a shared memory location and others providing a complete UI from which profiling values can be observed. Using a single aspect-based property throughout the application avoids these variations.

The independent development and subsequent composition of the attributed components supporting primary controller functionality and the profiling aspect-based property is made possible by language-independent AOP. Normally, crosscutting with AOP exposes language dependencies when aspects and components are woven. Composition can then go on if the components and aspects involved coordinate on the basis of language, but to do so takes away from the independence of their developers. With language-independent AOP, these components and aspects will be woven together, regardless of their implementation language. Figure 3.13 visualises the composition available with language-independent AOP that is exploited by the application integrator role. In this figure, attributed components 'A', 'B' and 'C' are composed into an application. Interoperability among components written in different languages is familiar to component-oriented programming. Aspect-based properties go further by allowing functionality to crosscut components of a system. This crosscutting corresponds to language-independent AOP, because the aspects and the components they crosscut can be implemented in a variety of languages. For example, in Figure 3.13 aspects in the form of aspect-based properties written in C#, SML.NET and Visual Basic crosscut three components also written in these languages. Here, the aspects provide persistence, security and logging functionality.

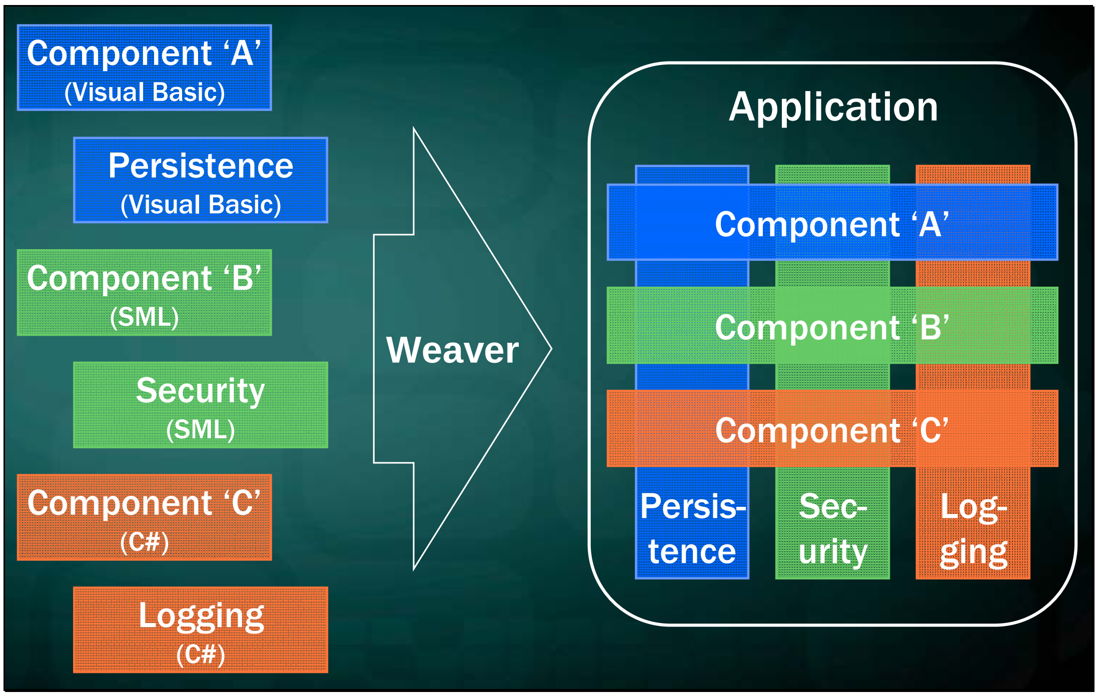


Figure 3.13: Overview of language-independent AOP.

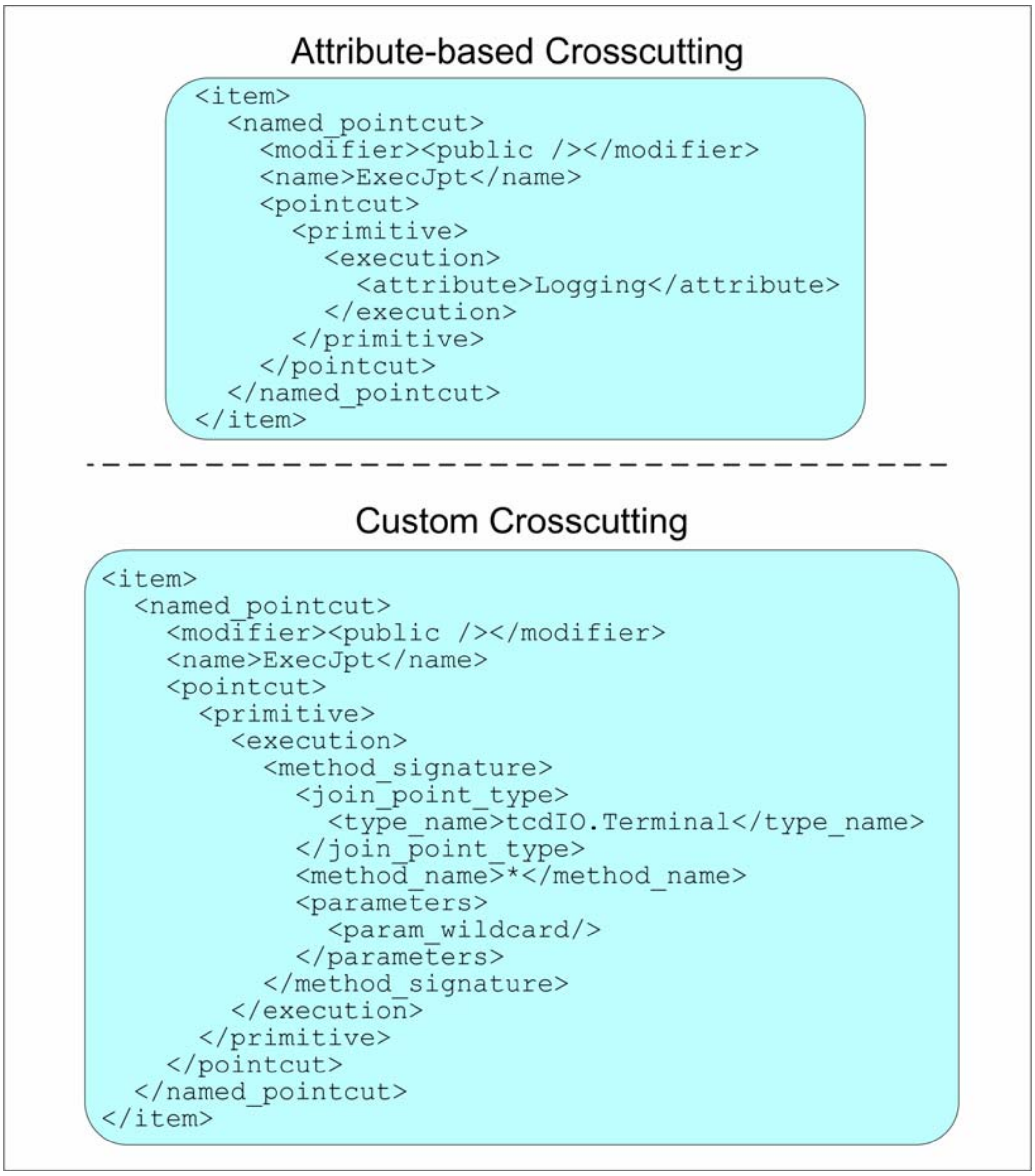
3.1.3.2 Using Custom Crosscutting

The previous scenario presented the ideal application of our programming model. The application integrator benefited greatly from the presence of attribute annotations in the components being integrated into an application. Using attribute types, the component implementer had been able to specify the bindings between the component and an aspect-based property implementing profiling. Thus, the application integrator did not need to understand the AOP mechanism and the operations it relied upon to implement crosscutting. More importantly, the attributed components and aspect-based property could be used 'as is'. Independently developed components could be crosscut without modification.

To deal with situations in which components are not properly annotated, the programming model makes available a customization mechanism in the form of custom crosscutting. Previously, we introduced the application integrator as being responsible for choosing required attributed components, aspect-based properties consistent with the attribute type accompanying the attributed component, and for creating integration components for a specific application. In addition to these functions, the application integrator role can customise the crosscutting specification of an existing aspect-based property. This involves making application-specific edits to the aspect-based property. Exercising this capability is extremely restrictive as it introduces application-specific dependencies into the crosscutting specification being modified. The justification for allowing such changes is that it allows the programming model to incorporate legacy components that cannot be dealt with by the component writer role. For example, component source may not be available or recompilation may involve using a new compiler that can potentially introduce new flaws. Moreover, custom crosscutting specifications allow our model to avoid preplanning problems and allow tailorability even for legacy components.

Custom crosscutting involves writing new pointcuts for an existing crosscutting specification. For example, we may want to modify the logging aspect-based property of Figure 3.7 so that it applies logging to the execution of methods of an existing I/O library such as `tcdIO` [Cah'02] that was not previously annotated with the `Logging` attribute. The updated pointcut appears in Figure 3.14 alongside the attributed-based original. The updated pointcut is similar to the original in that it makes use of a single execution primitive pointcut designator. The difference is that instead of parameterizing the primitive pointcut designator in terms of an attribute, custom crosscutting is based on type specifications such as type names and the signatures of type members. We will elaborate on the syntax of this example in section 3.2.2, when we discuss the syntax of pointcuts that aspect-based properties use to select join points.

Besides allowing legacy components to be crosscut, custom crosscutting deals with scenarios in which the attribute type of an attributed component is not specifically supported by an aspect-based property, but this aspect-based property provides the underlying functionality required to support the attribute type. Such a situation occurs when the attribute types used to annotate an attributed component differ in name to those expected by the aspect-based property, but not in terms of semantics. For example, different attribute types for logging may be satisfied by the same aspect-based property if



```
<item>
  <named_pointcut>
    <modifier><public /></modifier>
    <name>ExecJpt</name>
    <pointcut>
      <primitive>
        <execution>
          <attribute>Logging</attribute>
        </execution>
      </primitive>
    </pointcut>
  </named_pointcut>
</item>
```

```
<item>
  <named_pointcut>
    <modifier><public /></modifier>
    <name>ExecJpt</name>
    <pointcut>
      <primitive>
        <execution>
          <method_signature>
            <join_point_type>
              <type_name>tcdIO.Terminal</type_name>
            </join_point_type>
            <method_name>*</method_name>
            <parameters>
              <param_wildcard/>
            </parameters>
          </method_signature>
        </execution>
      </primitive>
    </pointcut>
  </named_pointcut>
</item>
```

Figure 3.14: Sample of custom crosscutting used to attributed-based crosscutting of Figure 3.7.

the attributes used to specify logging have different names but the same meaning. In such cases, custom crosscutting allows the application integrator to step in and integrate the attributed component and aspect-based property by amending an existing pointcut.

Custom crosscutting is loosely based on the abstract pointcut facility available with AspectJ [Asp'02] upon which we base our aspect model. In the AspectJ language specification, abstract pointcuts are used to make aspects reusable. Abstract pointcuts allow advice to be defined independently of a particular application in which this advice will be applied. Reuse is a matter of providing a concrete implementation for the abstract join point. Our model for customization differs slightly in that there is an existing pointcut that is being modified rather than there being no existing implementation.

3.1.4 Application Deployer

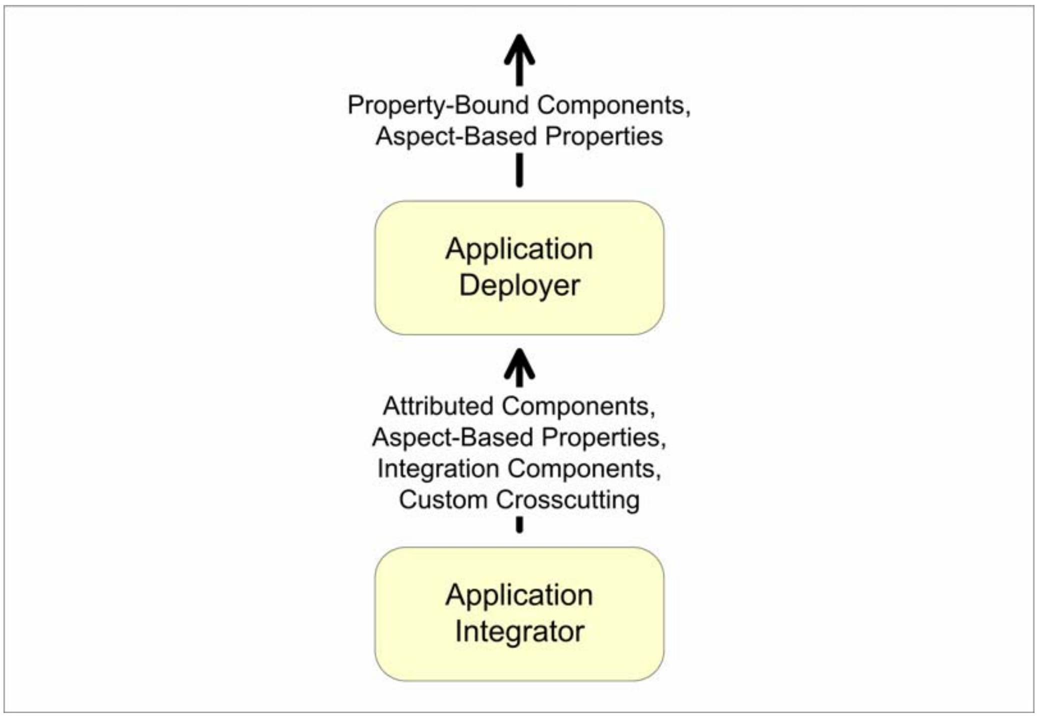


Figure 3.15: Overview of products required and produced by application deployer role.

The application deployer is responsible for ensuring that attributed components are woven with aspect-based properties. An overview of the products required and created by this role is shown in Figure 3.15. Essentially, the application deployer exploits a weaver to weave aspect-based properties into the components of an application. Weaving is carried out according to the crosscutting specifications of aspect-based properties, whether they are the original version created by the aspect-based property writer or versions with custom crosscutting added by the application integrator. A post-weave component is referred to as a property-bound component to indicate that any required crosscutting functionality has been bound to the component. All components produced by the application deployer are property-bound components, as the weaver is applied to all components of an application, regardless as to whether or not they are annotated with an attribute type. Access to attribute types is also required as the weaver does not remove attributes exploited by aspect-based properties from components, and so it may happen, for whatever reason, that attributes defined by an attribute type are inspected using the component platform's reflection API. Such introspection has been known to require that the attribute type be instantiated, and doing so requires the type implementation to be on hand.

Chapter 2 pointed out that load-time weaving was well suited to weaving components, as load-time weaving is consistent with the deployment characteristics of components and suits PA crosscutting semantics. In the case that the component distribution format includes a rich self-describing metadata and component behaviour is written in terms of byte code, load-time weaving also supports clear-box crosscutting. Clear-box crosscutting is required to support the finer granularity crosscutting available with a PA mechanism.

Also, load-time weaving meets our requirements for language-independence. Recall from Figure 3.13, that the weaver used to compose aspect-based properties with attributed components and other components will support language-independent AOP. With respect to weaving, language-independence requires weaving be based on details stored in the distribution format of the component rather than details in the source code used to implement the component. Load-time weaving suits language-independence as it makes exclusive use of the component's distribution format.

A variety of mechanisms are suitable for implementing load-time weaving; however, the weaver architecture used by aspect-based properties is modelled on the mechanism used by Binary Component Adaptation (BCA) [Coh'98]. The BCA mechanism was presented in the context of other weaving mechanisms in Figure 2.22. BCA distinguishes itself from other load-time transformers in that it uses a modified JVM in which the bootstrap loader is replaced with one that hooks into the BCA transformation engine. The programming model for aspect-based properties uses the same approach. Before invoking the weaver, the application deployer must make the weaver aware of the aspect-based properties that are available. In our prototype implementation, the weaver locates aspect-based properties by looking at a directory known to the weaver, but other implementations may provide a different means to identify aspect-based properties, for instance through configuration files or an operation system registry. In addition to correctly positioning aspect-based property files, the application deployer is responsible for launching the component execution environment using the modified bootstrap loader.

3.1.5 Programming Model Summary

Application development involving aspect-based properties is split between four roles shown together with their products in Figure 3.16. Reusability is addressed by splitting the implementation of aspect-based properties and their binding to a particular component between two programming roles. On the one hand, the aspect-based property writer produces aspect-based properties for use in an application. Aspect-based properties consist

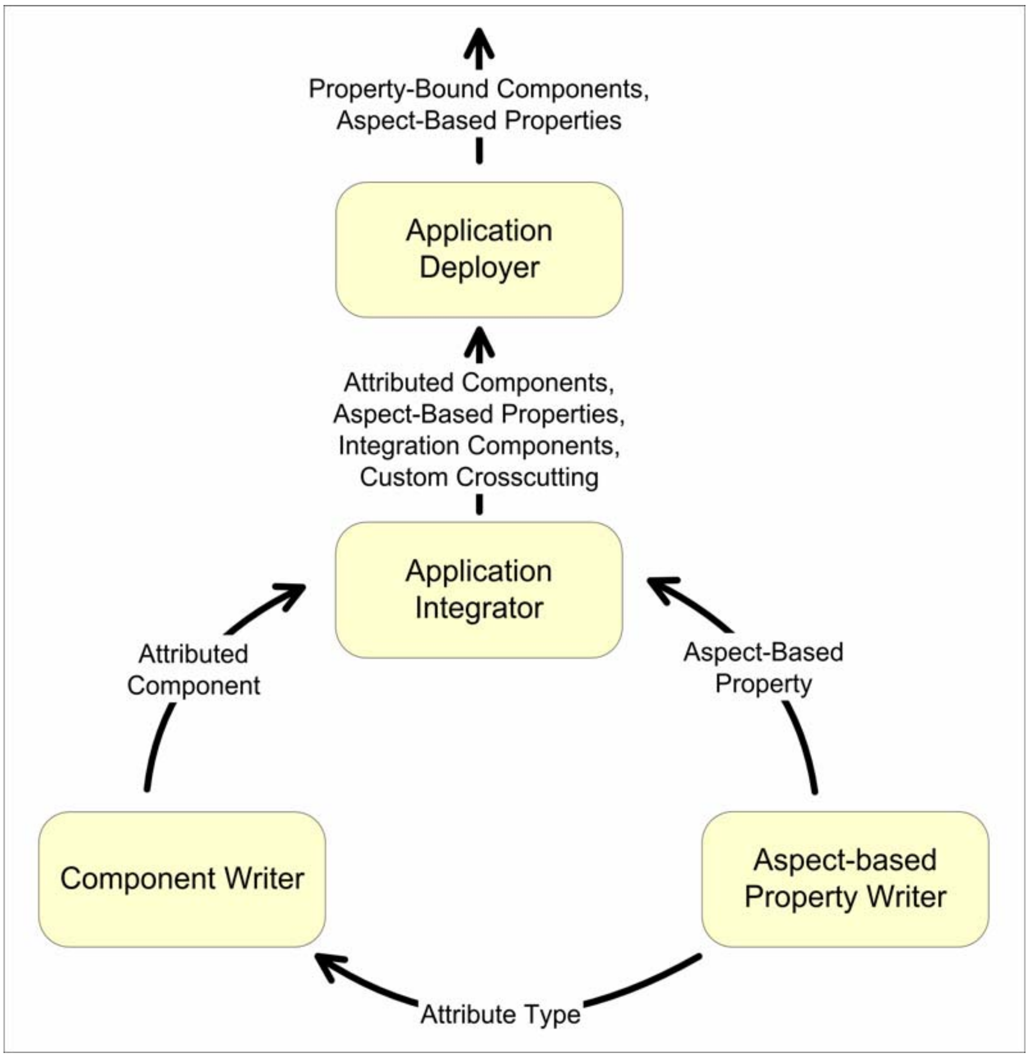


Figure 3.16: Overview of roles and products of development involving aspect-based properties.

of XML-based PA crosscutting specifications and a component implementing advice behaviour. Attribute-based crosscutting is used to decouple the aspect-based property and that component to which it will be applied, and an attribute type is provided for accessing aspect-based property functionality. Thus, any component writer is able to use an aspect-based property by annotating the component source in accordance with the attribute type. The annotated component is referred to as an attributed component, and it is distributed with the attribute types with which it is annotated. Attributed components and aspect-based properties are interoperable regardless of implementing language as they are woven together using language-independent AOP. Thus, an application integrator can select attributed components, and aspect-based properties without regard for their implementing language. The application integrator can exploit custom crosscutting to crosscut legacy components and to adapt aspect-based properties to non-native attribute types. Integration components required for application specific integration may also be produced by this role. All these products are passed to the application deployer, who ensures that the load-time weaver has access to all aspect-based properties, and launches the execution environment with the weaver's bootstrap loader.

3.2 Writing Crosscutting Semantics

Attribute-based crosscutting specifications of aspect-based properties as well as custom crosscutting specifications are written using a PA mechanism. To understand how to write a crosscutting specification, we discuss the task of writing a logging aspect-based property in terms of the aspect model available with this PA mechanism. In subsection 3.2.1, we look at where logging can be added to the execution of component code. The points available correspond to the join point model. In subsection 3.2.2, we look at how these join points are selected for logging. The means of identifying join points is through pointcuts. So, in this subsection, the primitive pointcut designators available are enumerated before a series of examples are provided to detail how a pointcut is written in XML. Certain types of logging will want to report execution parameters. These are accessed through typed formal parameters, and so we cover binding of join point execution context to typed formal parameters. Broadly speaking, three techniques exist for writing crosscuts with pointcuts, and these techniques can be placed in a spectrum according to how coupled the crosscut is to implementation of join points that it references. Custom crosscutting is realised with name-based crosscutting in which pointcuts are parameterised with precise signature and type information, or with property-based crosscutting in which uses signature and type patterns are used. The attribute-based crosscutting used with aspect-based properties relies on property-based crosscutting written in terms of attributes, which references no join point implementation. We provide overviews of each approach in turn. Logging behaviour is bound to pointcuts using advice, which is the means of modifying join point semantics. Advice is the focus of the final subsection. Essentially, advice specification is about cross-referencing the method implementing advice with the pointcut describing where it is applied. In this subsection, the XML schema for advice is presented along with a programming example of advice that accesses join point execution context and one that does not.

3.2.1 The Join Point Model

The points in program execution at which we can bind logging are dictated by the PA mechanism's join point model. With aspect-based properties, the join point model is that of AspectJ V1.0.6. The join point types available are listed in Table 3.1. This table is identical to Table 2.2, which presented the join points available in the AspectJ PA model.

Table 3.1: Join point types available with aspect-based properties.

Join point category	Join point types
Execution	Method execution Initializer execution Constructor execution Static initializer execution Handler execution Object initialization
Call	Method call Constructor call Object pre-initialization
Field access	Field reference Field assignment

Note that due to the language-independence constraints of the aspect-based property programming model, the join points identified in the join point model are not specific to a particular programming language. Rather, they are join points that are identifiable in the distribution format of the components for the component platform for which aspect-based properties are implemented. If the component format provides a discernable implementation for these join points, then it is straightforward to implement aspect-based properties for the component platform. Otherwise, additional work will be required to map the join points to whatever structures are discernable, and where a mapping is not available the full join point model may not be available.

The logging example considered in this section should report the execution of methods within a type so that stack traces detailing program execution can be observed from a logging file. This limits the join points of interest to method and constructor executions. Constructor execution join points offer finer grained tracking than object initialization join points. For simplicity, our example ignores initializer join points.

3.2.2 Means of Identifying Join Points

Selecting join points for logging is a matter of writing a pointcut. Pointcuts in aspect-based properties are not embedded in advice expressions as they are in AspectJ. Instead, advice is applied to a pointcut by referencing the name assigned to a named pointcut during its declaration. We borrow from AspectJ terminology when we refer to a pointcut defined in this manner as a *named pointcut* [Asp'02]. As well as its name, the declaration of a named pointcut includes a declaration of its typed formal parameters and a series of logically combined primitive pointcut designator specifications. The primitive pointcut designators available mirror the set available with the AspectJ V1.0.6 PA model. These designators are summarised in Table 3.2, and their semantics are consistent with their descriptions in section 2.2.1.

The need to always use named pointcuts defined in aspect-based properties may at first appear to be more cumbersome than simply including them as part of an advice statement. In fact, providing a clear separation between advice and pointcuts reduces the complexity of the XML schema for advice by eliminating the pointcut-related tags. Another advantage is that the separation limits custom crosscutting from having to modify advice.

Table 3.2: Primitive pointcut designators available with aspect-based properties.

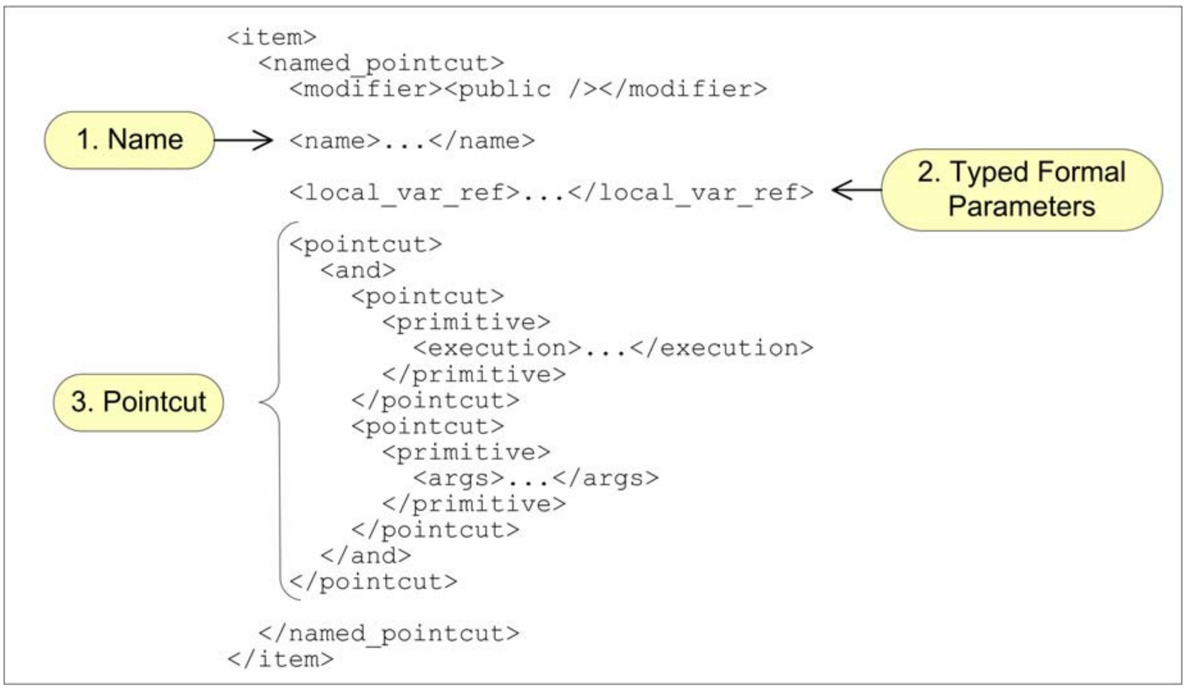
`call(`*`Signature`*`)`	`within(`*`TypePattern`*`)`
`execution(`*`Signature`*`)`	`withincode(`*`Signature`*`)`
`initialization(`*`Signature`*`)`	`this(`*`TypePattern or Id`*`)`
`get(`*`Signature`*`)`	`target(`*`TypePattern or Id`*`)`
`set(`*`Signature`*`)`	`args(`*`TypePattern or Id, ...`*`)`
`handler(`*`TypePattern`*`)`	`cflow(`*`pointcut`*`)`
`staticinitialization(`*`TypePattern`*`)`	`cflowbelow(`*`pointcut`*`)`

The logical operators available for combining primitive pointcut designators are the same as those available with the AspectJ model, and they appear in Table 3.3. Syntactically, they are either expressed as XML tags, in the case of logical *and* as well as logical *or*, or as an XML tag attribute in the case of logical not.

Table 3.3: Logical operations for combining primitive pointcut designators.

Operation	XML Expression	Semantics
`Or`	<or> <pointcut>…</pointcut> <pointcut>…</pointcut> </or>	Returns union of join points selected by primitive pointcut designators
`And`	<and> <pointcut>…</pointcut> <pointcut>…</pointcut> </and>	Returns intersection of join points selected by primitive pointcut designators
`Not`	<pointcut logical_not="false">…</pointcut>	Inverts the meaning of a pointcut specification. NB: Implemented as an XML tag attribute, and not a tag itself.

To express the crosscutting required for logging, we would need to write a named pointcut to select method and constructor execution join points. An overview of the XML tag used to define a named pointcut is shown in Figure 3.17. The variable elements are enumerated in the figure, and the first item of interest is the named pointcut's name. This name



```
<item>
  <named_pointcut>
    <modifier><public /></modifier>

    <name>...</name>

    <local_var_ref>...</local_var_ref>

    <pointcut>
      <and>
        <pointcut>
          <primitive>
            <execution>...</execution>
          </primitive>
        </pointcut>
        <pointcut>
          <primitive>
            <args>...</args>
          </primitive>
        </pointcut>
      </and>
    </pointcut>

  </named_pointcut>
</item>
```

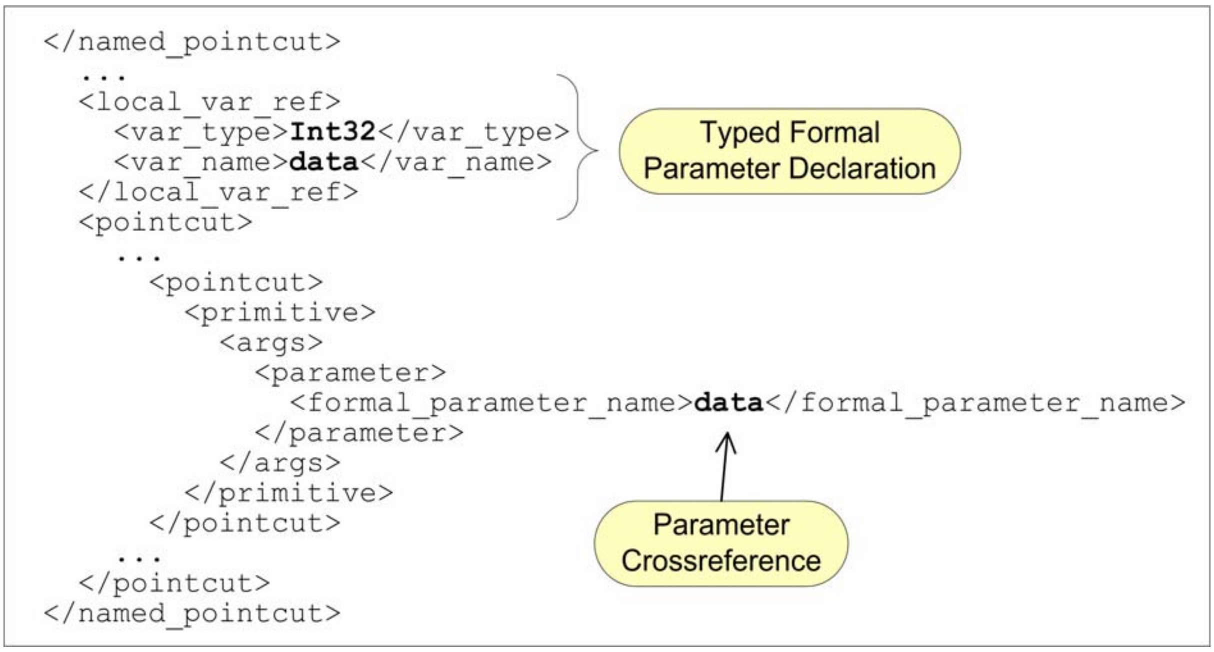
Figure 3.17: Overview of XML-based declaration of a named pointcut.

corresponds to a valid Java identifier, which reflects the Java origins of the AspectJ language specification. Implementations of aspect-based properties have the option of choosing a different identifier specification if required.

Logging provides richer information if details of join point execution state are recorded. Access to join point execution context is through typed formal parameters defined in the pointcut specification. All typed formal parameters are first declared in the `<local_var_ref>` tag. Each declaration specifies the variable's type and name. The typed formal parameter is bound to a variable in the context of join point execution. This binding occurs when the typed formal parameter's name is referenced by context-exposing primitive pointcut designators. These are the `this`, `target`, and `args` primitive pointcut designators, which are described in detail in Table 2.4 on page 36, and we will see an example of their use in the next figure.

The final element of the named pointcut contains the primitive pointcut designators that bind typed formal parameters, and select join points. In Figure 3.17, two primitive pointcut designators are combined using the logical *and* operator. The `args` primitive pointcut designator is used to bind the typed formal parameter, while the `execution` primitive pointcut designator is used to select execution join points. For brevity, their inner tags have been truncated.

The specifics of using typed formal parameters appear in Figure 3.18, where a typed formal parameter is declared and used in a primitive pointcut designator. The typed formal parameter is referenced by name in the primitive pointcut designator's arguments. Since this identifier is not used outside the crosscutting specification, we adopt the same naming conventions as AspectJ, and allow the name to be any valid Java-style identifier. However, the type used in the typed formal parameter declaration must be a valid type in the component platform for which aspect-based properties are being implemented. For example, our prototype implementation targets the CLI, and so the type name must be a valid CLI type name. For instance, the example in Figure 3.18 declares a typed formal parameter of the type `Int32` that is referenced by the name `data`.



```
</named_pointcut>
  ...
  <local_var_ref>
    <var_type>Int32</var_type>
    <var_name>data</var_name>
  </local_var_ref>
  <pointcut>
    ...
      <pointcut>
        <primitive>
          <args>
            <parameter>
              <formal_parameter_name>data</formal_parameter_name>
            </parameter>
          </args>
        </primitive>
      </pointcut>
    ...
  </pointcut>
</named_pointcut>
```

Figure 3.18: Overview of XML-based declaration and binding of typed formal parameters.

The ability to use typed formal parameters is limited by compilation dependencies introduced by their use. To pass a typed formal parameter to the aspect-based property's advice implementation requires that the method implementing advice have a parameter with the same type as the typed formal parameter. This requirement makes compilation of the method implementing advice dependent on being able to access a declaration for the typed formal parameter's type. Given that the intent of aspect-based properties is that they be developed separately from the components to which they are applied, it is hardly likely that declarations of specialised types will be available to aspect-based properties. To allow a reference to be obtained to objects whose type the method implementing advice has no knowledge of, the AspectJ specification allows a typed formal parameter to be declared to be of type `object`. Typed formal parameters of type `object` match any join point parameter type. Making use of this loophole with aspect-based properties is a matter of finding a type in the component platform targeted that has the same properties as type `object` in Java. For instance, the `Object` type in the CLI's unified type system offers this functionality, as any value type or reference type can be cast to `Object` a reference. Thus, it would be possible to replace `Int32` with `Object` in Figure 3.18.

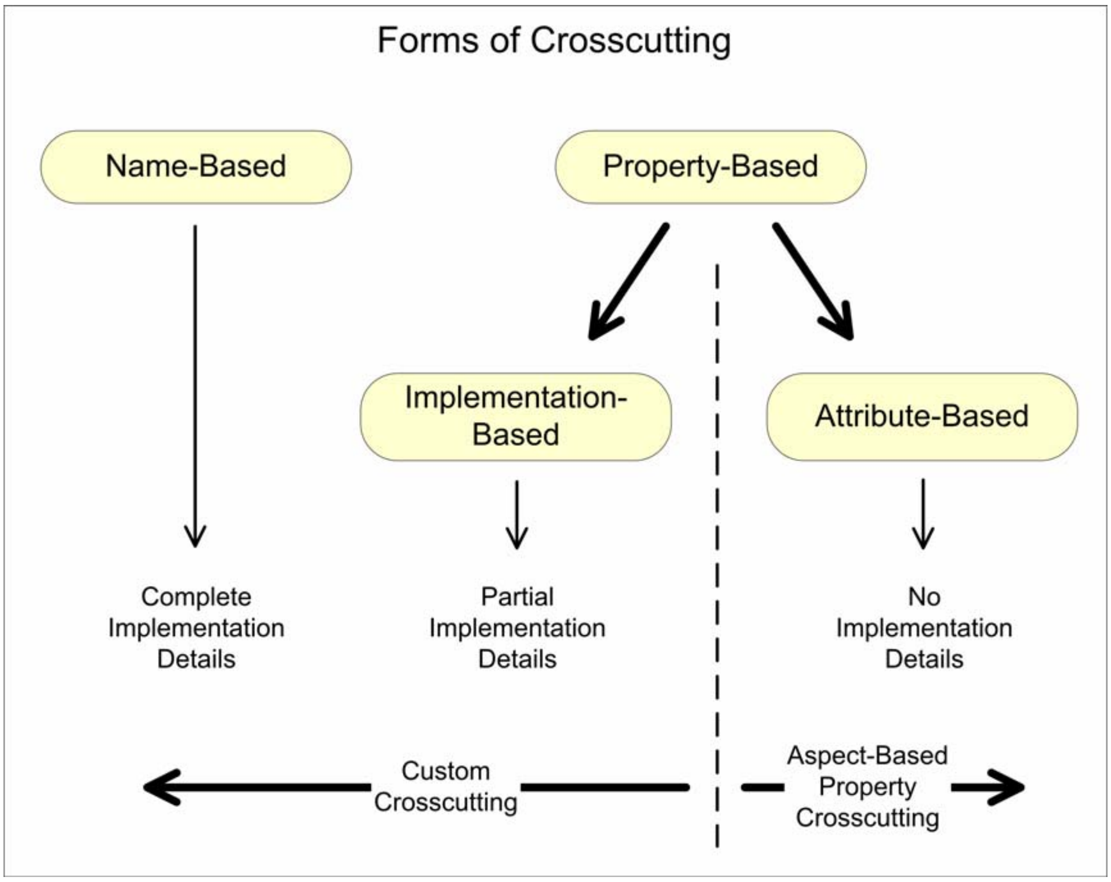
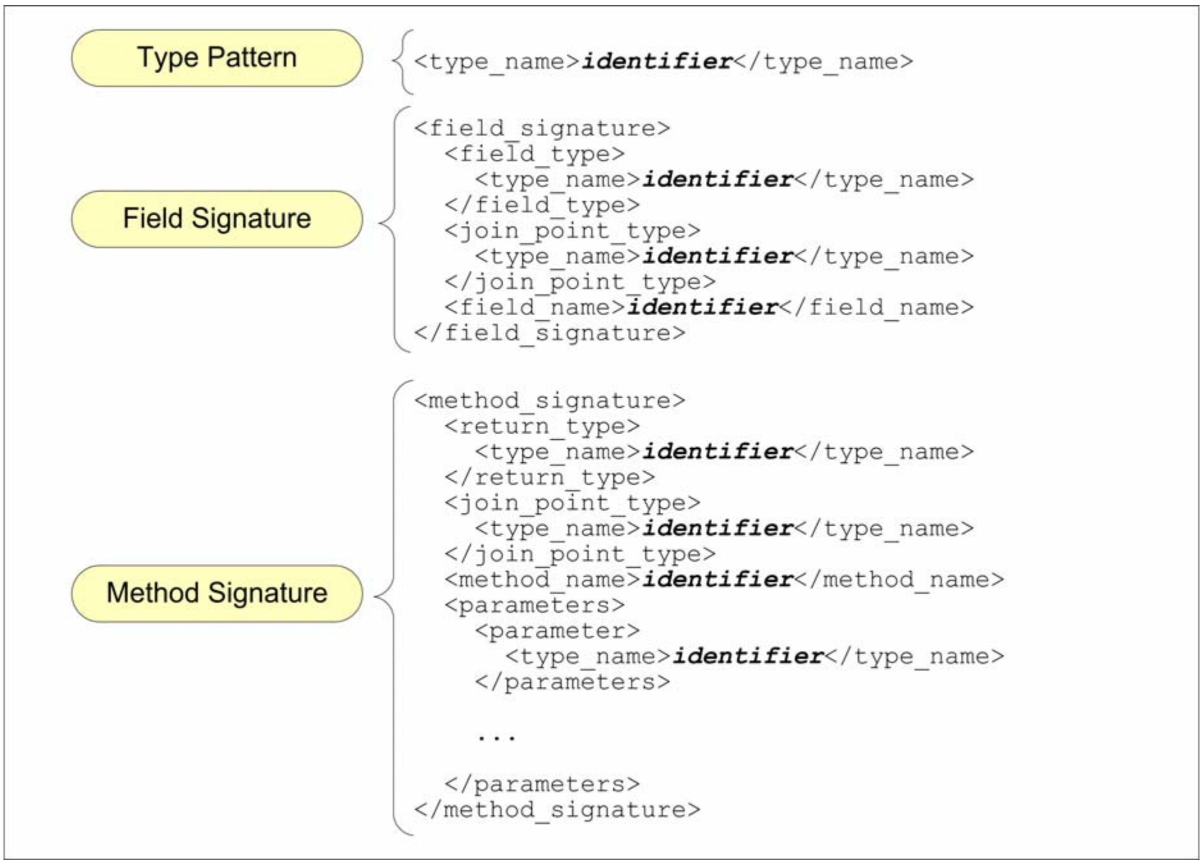

Forms of Crosscutting
Name-Based
Property-Based
Implementation-Based
Attribute-Based
Complete Implementation Details
Partial Implementation Details
No Implementation Details
Custom Crosscutting
Aspect-Based Property Crosscutting

Figure 3.19: Categorization of crosscutting specification according to implementation dependency.

In section 3.1, we explained that the attribute-based crosscutting written by an aspect-based property writer corresponds to property-based crosscuts written in terms of attributes. We also pointed out that custom crosscutting created by an application integrator uses name-based and property-based crosscutting written in terms of implementation details such as type specifications, method signatures and field signatures. These three approaches to crosscutting are placed along a continuum in Figure 3.19 from most to least implementation specific, and the figure draws a line between the approach taken to crosscutting when writing aspect-based properties and the approach taken when writing custom crosscutting. Specifically, the attribute-based crosscutting used with aspect-based properties avoids any reference to implementation details, whilst forms of crosscutting include partial or complete implementation details when identifying join points.

In the following subsections we describe the XML syntax for writing each of these three approaches.

3.2.2.1 Name-based Crosscutting



```
<type_name>identifier</type_name>
```

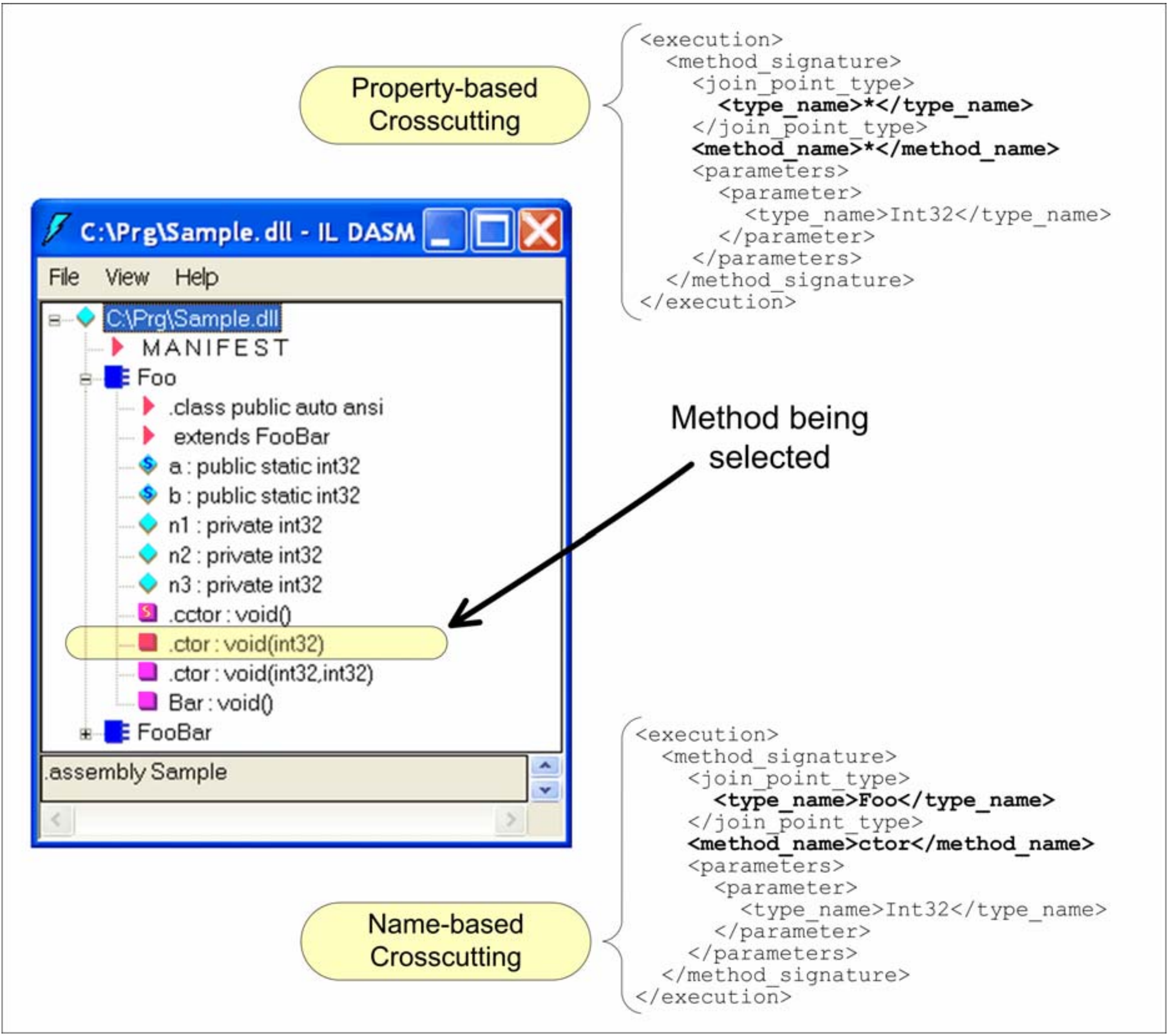
```
<field_signature>
  <field_type>
    <type_name>identifier</type_name>
  </field_type>
  <join_point_type>
    <type_name>identifier</type_name>
  </join_point_type>
  <field_name>identifier</field_name>
</field_signature>
```

```
<method_signature>
  <return_type>
    <type_name>identifier</type_name>
  </return_type>
  <join_point_type>
    <type_name>identifier</type_name>
  </join_point_type>
  <method_name>identifier</method_name>
  <parameters>
    <parameter>
      <type_name>identifier</type_name>
    </parameters>

    ...

  </parameters>
</method_signature>
```

Figure 3.20: Overview of XML type patterns and signatures used for name-based crosscutting.

Name-based crosscutting occurs when a primitive pointcut designator's arguments are precise metadata descriptions. Several primitive pointcut designators in Table 3.2 take signature or type arguments. The general format for XML specifying these arguments is shown in Figure 3.20, where `identifier` is a string that corresponds either to a type defined for the component platform for which aspect-based properties are implemented or to the name of a field or method. With name-based crosscutting, type pattern arguments are specified with a specific type name, which takes the place of `identifier`. Signatures are used for identifying fields and methods. Here, the `<parameter>` tags correspond to the arguments in a method signature, so the number of these tags used in a signature varies. The `<join_point_type>` tag identifies the type containing the field or method being identified. Note that constructors correspond to methods for which the `<return_type>` tag is missing and for which the method name is "`ctor`". Name-based crosscutting requires that the details in the pointcut specification match exactly the metadata description of the structure implementing join points that the pointcut is selecting. Metadata details do not necessarily correspond to the programming language specification for a signature or type. Take the example of the CLI, which is targeted by our prototype. The metadata type `System.Int32` corresponds to the `int` type in C# and `Integer` in VisualBasic.NET.

When custom crosscuts are being written, it is useful to examine the software component's metadata to determine the exact signature rather than rely on the written API of a component, which may specify types and signatures in terms of source code. For instance, CLI provides the Ildasm program with introspection facilities that allow component metadata to be examined directly. This program can be used to dissect metadata details from a component to ensure that name-based crosscutting is written correctly. For example, in Figure 3.5 of section 3.1.1.1, Ildasm was referenced when name-based crosscutting was written to capture constructor join points corresponding to the execution of a particular constructor. For reference, this example appears again at the bottom of Figure 3.21.



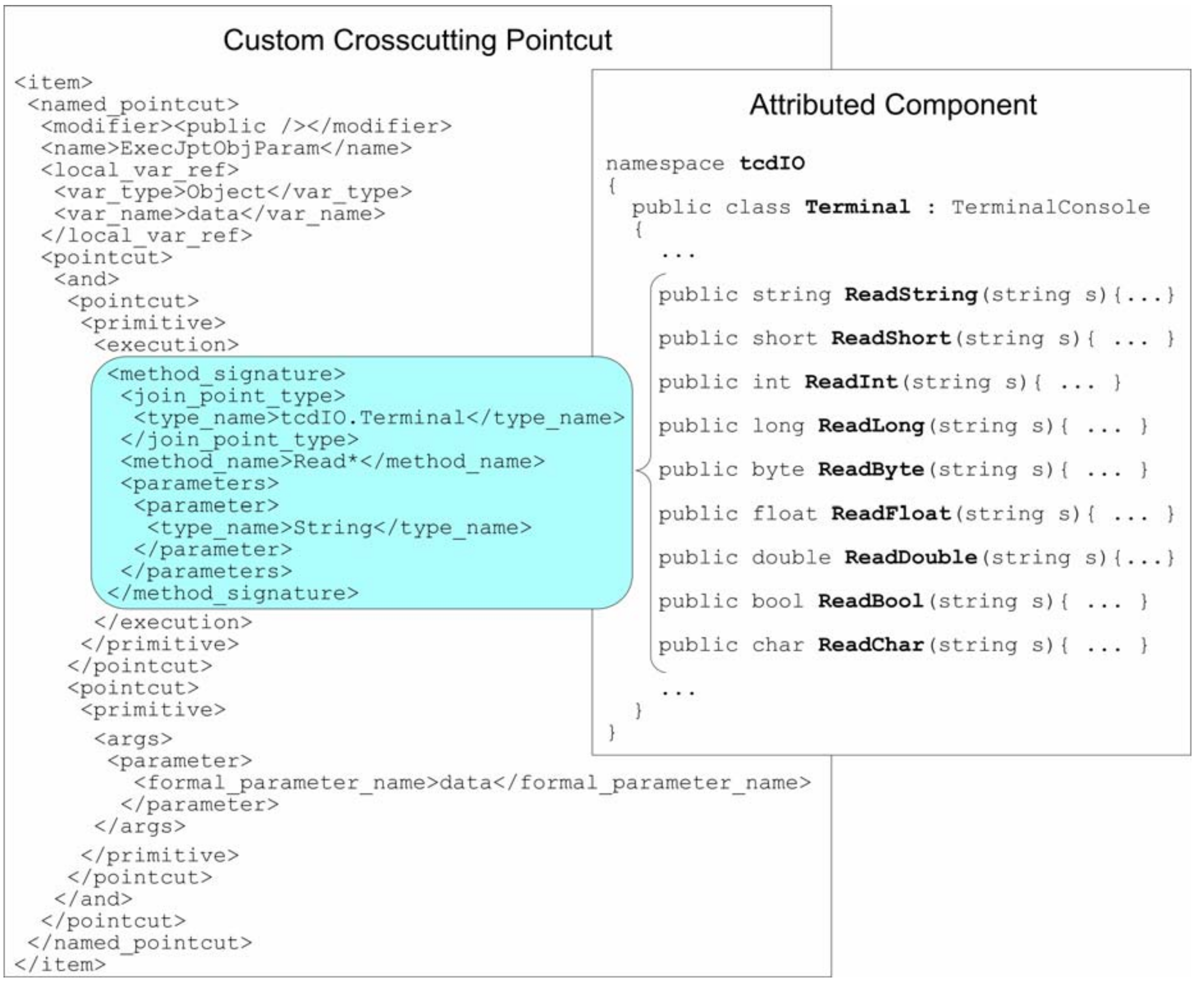
```
<execution>
  <method_signature>
    <join_point_type>
      <type_name>*</type_name>
    </join_point_type>
    <method_name>*</method_name>
    <parameters>
      <parameter>
        <type_name>Int32</type_name>
      </parameter>
    </parameters>
  </method_signature>
</execution>
```

```
<execution>
  <method_signature>
    <join_point_type>
      <type_name>Foo</type_name>
    </join_point_type>
    <method_name>ctor</method_name>
    <parameters>
      <parameter>
        <type_name>Int32</type_name>
      </parameter>
    </parameters>
  </method_signature>
</execution>
```

Figure 3.21: Name-based and property-based alternatives for selecting join points corresponding to the execution of a `class Foo` constructor with a single argument.

3.2.2.2 Implementation-based Property-based Crosscutting

Implementation-based property-based crosscutting relies on partial descriptions of signatures and type patterns rather complete signatures and specific type names. We can think of such property-based crosscutting as specifying signatures and type patterns with regular expressions rather than precise strings. At the top of Figure 3.21, is a property-based alternative to name-based crosscutting in which most identifiers are specified using wildcard characters. This example demonstrates the use of '`*`' in the XML, which is a wildcard character that allows identifiers and type names to be specified with a regular expression. '`*`' can be used at any point in an identifier, and it matches any combination of characters. When it comes to parameters, a more powerful wildcard is available in the form of the `<param_wildcard>` tag, which is used in place of a set of `<parameter>` tags. `<param_wildcard>` matches any combination of parameters, regardless of type or number. Finally, the specification of type patterns need not be done with a single `<type_name>` tag. Rather, several `<type_name>` tags can be combined using the logical operator tags of Table 3.3. When doing so, `<type_name>` tags appear in place of `<pointcut>` tags shown in the table.



```
<item>
 <named_pointcut>
  <modifier><public /></modifier>
  <name>ExecJptObjParam</name>
  <local_var_ref>
   <var_type>Object</var_type>
   <var_name>data</var_name>
  </local_var_ref>
  <pointcut>
   <and>
    <pointcut>
     <primitive>
      <execution>
       <method_signature>
        <join_point_type>
         <type_name>tcdIO.Terminal</type_name>
        </join_point_type>
        <method_name>Read*</method_name>
        <parameters>
         <parameter>
          <type_name>String</type_name>
         </parameter>
        </parameters>
       </method_signature>
      </execution>
     </primitive>
    </pointcut>
    <pointcut>
     <primitive>
      <args>
       <parameter>
         <formal_parameter_name>data</formal_parameter_name>
        </parameter>
      </args>
     </primitive>
    </pointcut>
   </and>
  </pointcut>
 </named_pointcut>
</item>
```

```
namespace tcdIO
{
  public class Terminal : TerminalConsole
  {
    ...
    public string ReadString(string s){...}
    public short ReadShort(string s){ ... }
    public int ReadInt(string s){ ... }
    public long ReadLong(string s){ ... }
    public byte ReadByte(string s){ ... }
    public float ReadFloat(string s){ ... }
    public double ReadDouble(string s){...}
    public bool ReadBool(string s){ ... }
    public char ReadChar(string s){ ... }
    ...
  }
}
```

Figure 3.22: Sample of custom crosscutting with implementation-based property-based crosscutting.

The pointcut in Figure 3.22 is typical of custom crosscutting that uses implementation-based property-based crosscutting to apply aspect-based property functionality to a legacy component. This example includes a typed formal parameter of type `Object` that is assigned a reference to the first parameter of the method matching the pointcut. Implicitly, this means the matching method must have at least one parameter. Of more importance, as far as custom crosscutting is concerned, is that this example uses the containing type name and a method prefix to crosscut the execution of several methods of a legacy component rather than the specific signatures of each method. The component being crosscut is assumed to be a legacy component in that its implementation cannot be revised to include the attributes required to bind to the logging aspect-based property's original crosscutting specification. An important assumption in using custom crosscutting is that the application integrator writing the custom crosscutting will have sufficient knowledge of the component being crosscut to select correct join points and not to interfere with correct operation of the application being crosscut.

3.2.2.3 Attribute-based Property-based Crosscutting

When an aspect-based property writer writes the crosscutting specification of aspect-based properties, as opposed to the custom crosscutting implemented by an application integrator, attributed-based crosscutting should be used. The syntax for writing such pointcuts is derived by substituting type pattern or signature parameters shown in Figure 3.20 with `<attribute>` tag shown in Figure 3.23.

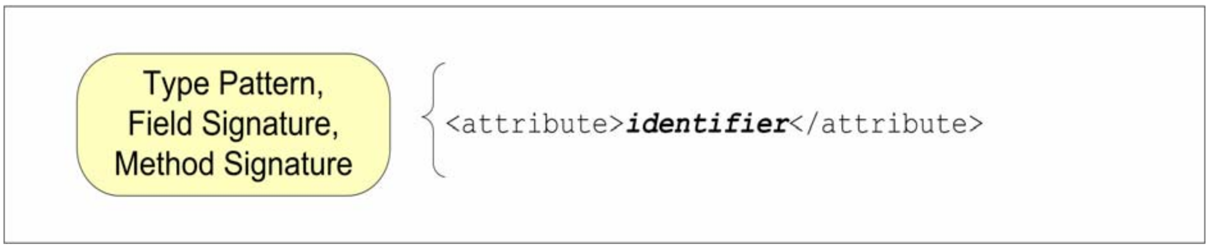
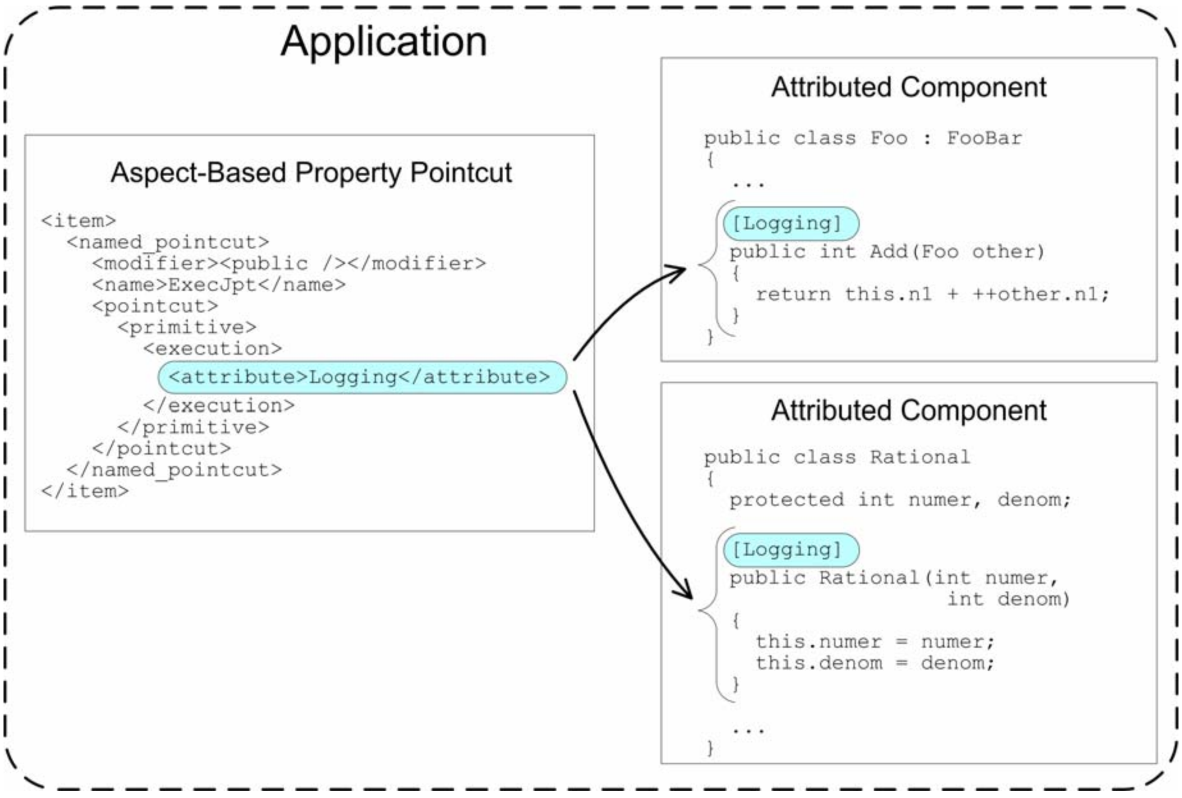


Figure 3.23: XML used to specify signatures or type patterns in terms of attribute types.

An example of attribute-based property-based crosscutting appears in the pointcut of Figure 3.24. This pointcut is reminiscent of the crosscutting specifications presented in the overview of aspect-based properties in Figure 3.7, page 71, in that the pointcut specification is the same. However, Figure 3.24 puts the pointcut in the context of an overall application in which multiple attributed components are being crosscut. The example underscores the point that writing the pointcut in term of attributes allows the pointcut to select join points in a variety of attributed components without regard for their implementation specifics. In the figure, methods from two attributed components are selected for logging based on annotation of their implementation with the `Logging` attribute. Recall that aspect-based property writer should complement the pointcuts that reference attributes with attribute type implementations that allow the component writer to annotate component code. These attribute types provide an API by which aspect-based property functionality is accessed. The name of an attribute type includes a namespace or package details that correspond to the aspect-based property being accessed as opposed to other aspect-based properties implementing logging. In Figure 3.24 the namespace details have been removed from the example for brevity. We provided a sample attribute implementation in Figure 3.8, but further details on implementation of attribute types for our prototype appear in Chapter 4.



```
<item>
  <named_pointcut>
    <modifier><public /></modifier>
    <name>ExecJpt</name>
    <pointcut>
      <primitive>
        <execution>
          <attribute>Logging</attribute>
        </execution>
      </primitive>
    </pointcut>
  </named_pointcut>
</item>
```

```
public class Foo : FooBar
{
  ...
  [Logging]
  public int Add(Foo other)
  {
    return this.n1 + ++other.n1;
  }
}
```

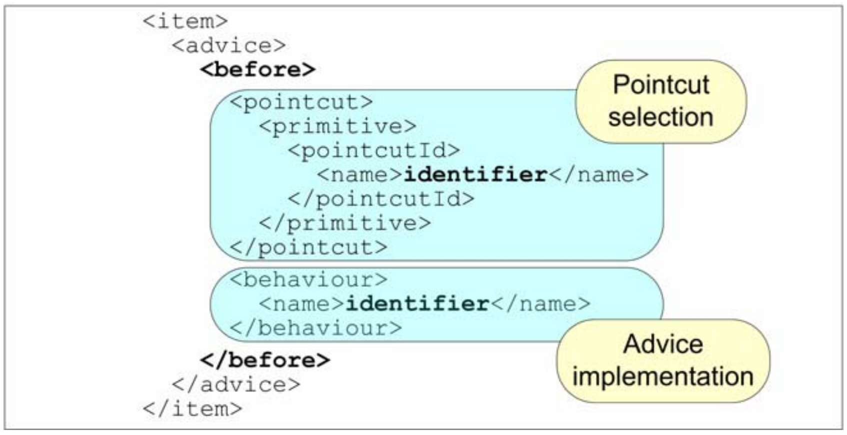
```
public class Rational
{
  protected int numer, denom;

  [Logging]
  public Rational(int numer,
                  int denom)
  {
    this.numer = numer;
    this.denom = denom;
  }

  ...
}
```

Figure 3.24: Attribute-based crosscutting used with aspect-based properties.

3.2.3 Means of Modifying Join Point Semantics

Having shown method execution join points being for logging with both custom crosscutting and attribute-based crosscutting, we need to add the logging functionality to the join points selected. Join point behaviour is modified using advice. The kinds of advice available with aspect-based properties are the same as those available with AspectJ, but the specification of advice with aspect-based properties is divorced from the implementation of advice whereas AspectJ couples the two. Thus, advice written for aspect-based properties does not have access to a reflective API or the ability to proceed from advice to join points via the keywords that are available with AspectJ. This decision reflects our interest in maintaining language-independence. Providing AspectJ-like language keywords would require extensions to the languages in which aspect-based property behaviour was implemented. To avoid this, Aspect-based properties make reflective and `proceed` functionality available through an interface.

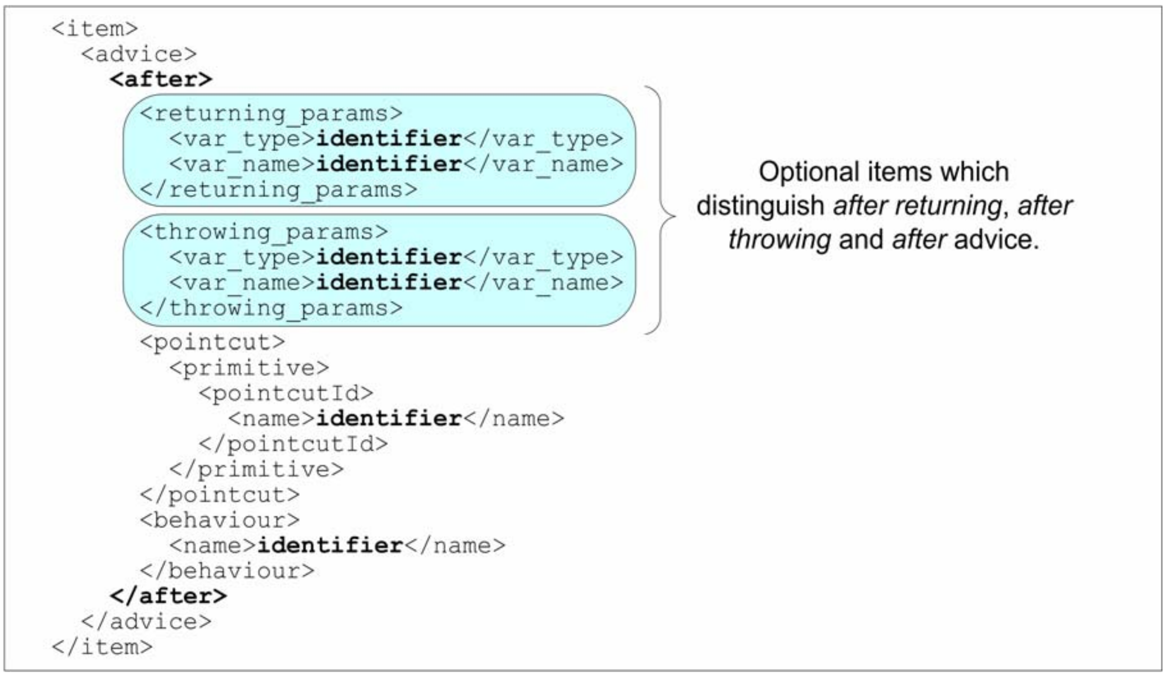
Join point behaviour can be modified using before, around and all flavours of after advice described in section 2.2.1. As with pointcuts, advice specifications are written in XML, and they cross-reference a named pointcut that selects the join points to which advice is applied with a method that implements the advice. The use of before and all types of after advice places no inheritance restrictions on the type implementing advice behaviour; however, this type must be instantiable through a default constructor (i.e., one without parameters). The instantiation requirement reflects the fact that a singleton [Gam'94] of the type implementing aspect-based property behaviour is instantiated at runtime. The instantiation of aspect types does not figure prominently in characterisations of the PA mechanism such as [Mas'03], but it is none the less useful as it allows aspects to have state. The implementation of around advice requires the ability to proceed to join point execution, and this functionality is provided by an API implemented by the weaver that weaves aspect-based properties and components. This API is accessed through inheritance. We will discuss this API further when we present an example implementation of around advice later in this section.



```
<item>
  <advice>
    <before>
      <pointcut>
        <primitive>
          <pointcutId>
            <name>identifier</name>
          </pointcutId>
        </primitive>
      </pointcut>
      <behaviour>
        <name>identifier</name>
      </behaviour>
    </before>
  </advice>
</item>
```

Figure 3.25: XML syntax for before advice.

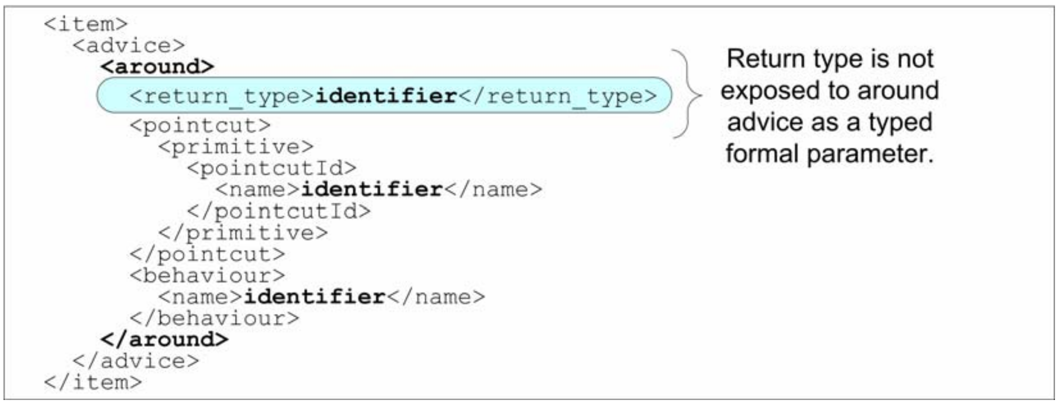
The schema used for advice specification varies according to the type of advice being specified. Overviews of the schemas for before, after and around advice are provided in Figure 3.25, Figure 3.26, and Figure 3.27 respectively. All schemas are similar in that they use the same syntax to reference advice behaviour as well as the pointcut that they affect. Essentially, references to the pointcut name and the method implementing advice behaviour are identifiers that cannot be specified with wildcard characters. Before, after, and around advice distinguish themselves by varying the tag containing the advice specifications according to the advice type, and the tags used are `<before>`, `<after>` and `<around>` respectively. These tags appear in bold in Figure 3.25, Figure 3.26, and Figure 3.27. In addition to pointcut and advice implementation details, around and after advice require additional information, which is specified using additional inner tags. The optional tags for after advice that are pointed out in Figure 3.26 allow after, after returning and after throwing advice to be distinguished. For after returning advice, a typed formal parameter must be declared to correspond to the value returned by the join point. This occurs inside a `<returning_params>` tag and involves specifying the type and name of the typed formal parameter. Being a typed formal parameter, a reference to the return value is passed to the method implementing advice, and this reference corresponds to the last advice method parameter. For join points that do not return a value, such as field set join points, the typed formal parameter should be declared with a type of `Void`. An almost identical approach is taken for specifying after throwing advice. The only difference is that the typed formal parameter must be declared inside a `<throwing_params>` XML tag so as to distinguish itself from after returning advice. Note that declaring the typed formal parameter to be of type `Object` will match any non-void return type. As with typed formal parameters declared in pointcuts, using a return type of `Object` will match any return type, and an `object` reference or its equivalent will be passed to the method advice implementing advice. Finally, after advice is specified when no typed formal parameter corresponding to the return or thrown type is declared.



```
<item>
  <advice>
    <after>
      <returning_params>
        <var_type>identifier</var_type>
        <var_name>identifier</var_name>
      </returning_params>
      <throwing_params>
        <var_type>identifier</var_type>
        <var_name>identifier</var_name>
      </throwing_params>
      <pointcut>
        <primitive>
          <pointcutId>
            <name>identifier</name>
          </pointcutId>
        </primitive>
      </pointcut>
      <behaviour>
        <name>identifier</name>
      </behaviour>
    </after>
  </advice>
</item>
```

Optional items which distinguish *after returning*, *after throwing* and *after* advice.

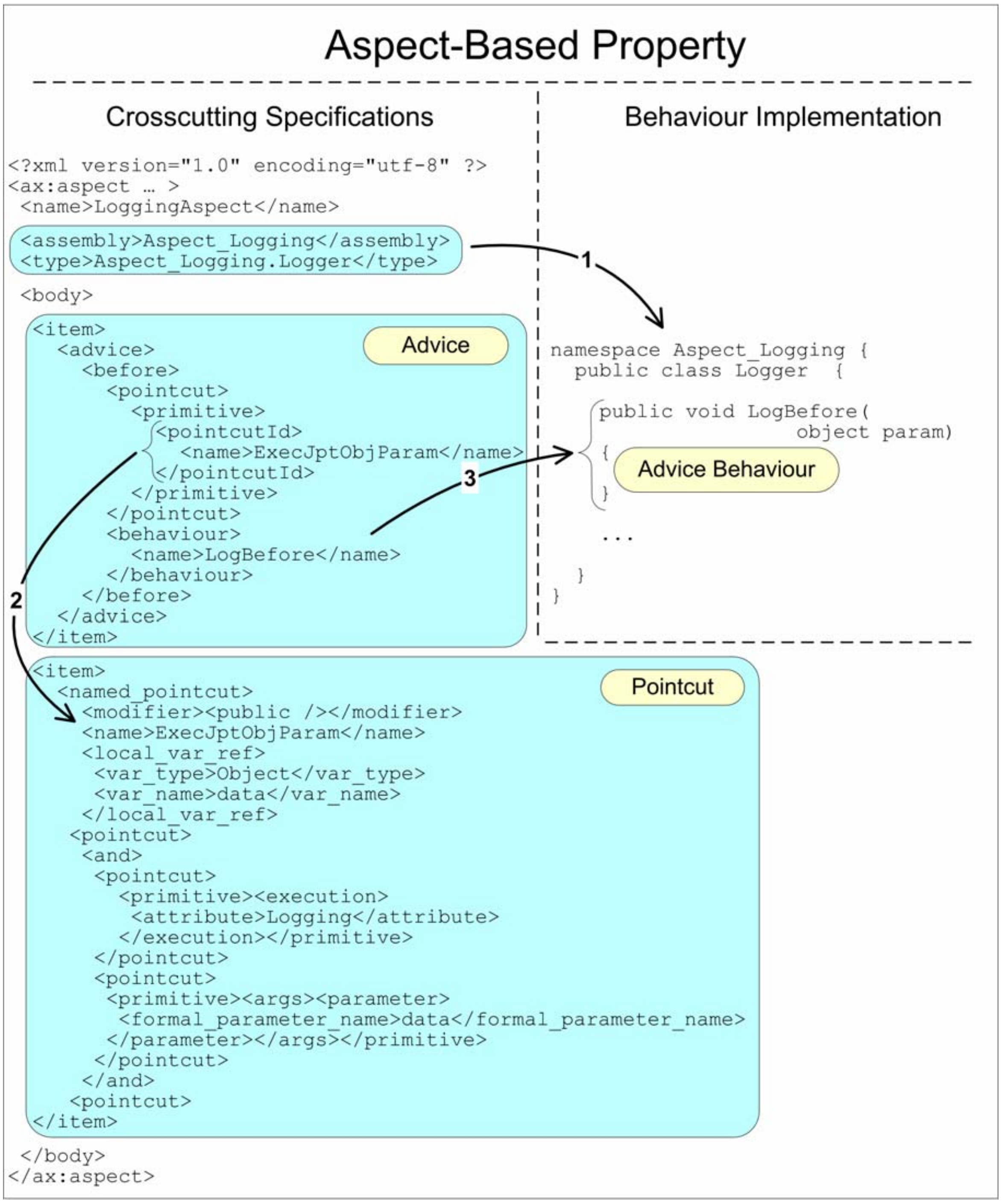
Figure 3.26: XML syntax for the variations of after advice.



```
<item>
  <advice>
    <around>
      <return_type>identifier</return_type>
      <pointcut>
        <primitive>
          <pointcutId>
            <name>identifier</name>
          </pointcutId>
        </primitive>
      </pointcut>
      <behaviour>
        <name>identifier</name>
      </behaviour>
    </around>
  </advice>
</item>
```

Return type is not exposed to around advice as a typed formal parameter.

Figure 3.27: XML syntax for around advice.



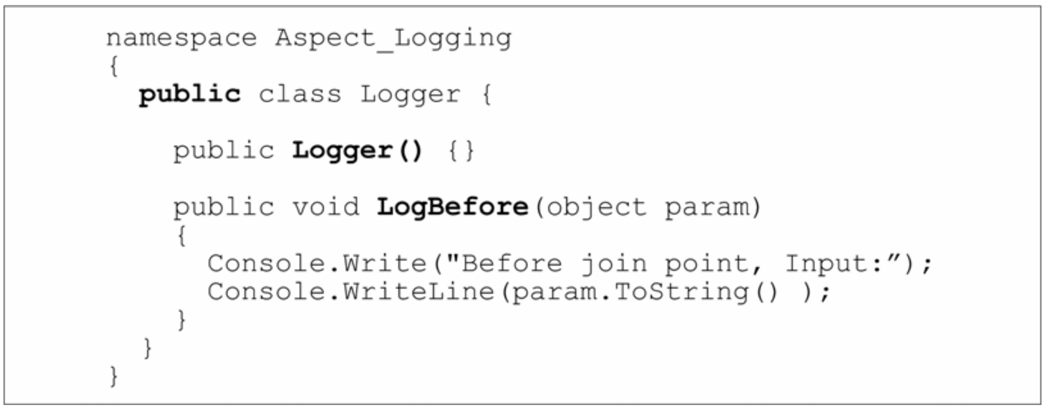
```
<?xml version="1.0" encoding="utf-8" ?>
<ax:aspect … >
 <name>LoggingAspect</name>
 <assembly>Aspect_Logging</assembly>
 <type>Aspect_Logging.Logger</type>
 <body>
  <item>
    <advice>
      <before>
        <pointcut>
          <primitive>
            <pointcutId>
              <name>ExecJptObjParam</name>
            </pointcutId>
          </primitive>
        </pointcut>
        <behaviour>
          <name>LogBefore</name>
        </behaviour>
      </before>
    </advice>
  </item>
  <item>
    <named_pointcut>
      <modifier><public /></modifier>
      <name>ExecJptObjParam</name>
      <local_var_ref>
       <var_type>Object</var_type>
       <var_name>data</var_name>
      </local_var_ref>
     <pointcut>
      <and>
       <pointcut>
         <primitive><execution>
          <attribute>Logging</attribute>
         </execution></primitive>
       </pointcut>
       <pointcut>
        <primitive><args><parameter>
         <formal_parameter_name>data</formal_parameter_name>
        </parameter></args></primitive>
       </pointcut>
      </and>
     <pointcut>
  </item>
 </body>
</ax:aspect>
```

```
namespace Aspect_Logging {
  public class Logger  {

    public void LogBefore(
                    object param)
    {
    }

    ...

  }
}
```

Figure 3.28: Example of before advice in the context of a complete aspect-based property.

In contrast to after advice, around advice specifications do not use a typed formal parameter to specify the return type. As pointed out in Figure 3.27, around advice simply includes a description of the return type of the join points to which it is applied. This subtle difference occurs as a join point result must be known for it to be assigned to a typed formal parameter. This is not possible in the case of around advice, as the around advice executes before the join point to which it is applied. Thus, a typed formal parameter cannot be used to specify the return type. The return type must still be known for correct join point matching, and so it is expressed directly. For consistency with typed formal parameters, the use of the `Object` type as the return type will match any return type.

The aspect-based property in Figure 3.28 includes a sample advice statement that maps the `ExecJptObjParam` pointcut to an advice implementation that creates a log entry when a join point is about to execute. We first introduced this named pointcut in Figure 3.22, where it was presented as an example of custom crosscutting in which a pointcut was tailored to a legacy component, but in this version attribute-based crosscutting is used. Let us first focus on the outer tags that encompass pointcut and advice statements, including those used to select the type implementing advice. Crosscutting specifications appear within an `<aspect>` XML tag, which contains leading `<assembly>` and `<type>` XML tags to identify the component and type containing the methods implementing advice behaviour. All methods implementing advice are contained in a single type to simplify support for aspect instantiation. By using a single type, aspect instantiation can be mapped to object instantiation in the underlying component platform. In Figure 3.28, these component and type references select a `class Logger` that happens to be implemented in C# as indicated by arrow ‘1’. Following these references is a `<body>` tag containing all advice and pointcut specifications for the aspect-based property. Recall that advice specifications indicate the type of advice being applied with a tag corresponding to the advice type, so the before advice in Figure 3.28 is specified within the `<before>` tag. This advice cross references a pointcut to which advice is applied. In this case, the `ExecJptObjParam` named pointcut is used, and this reference is indicated by arrow ‘2’. The `ExecJptObjParam` pointcut uses the execution primitive pointcut designator to limit the join points being logged to methods and constructors annotated with an attribute with the name `Logging`. The set of join points selected is further limited to those with at least one calling parameter by the use of the `args` primitive pointcut designator. The pointcut declares a single typed formal parameter of type `Object`, which the `args` primitive pointcut designator binds to the first parameter used to invoke the join point. The final element of the before advice is a reference to the method implementing advice. Advice behaviour is implemented by the `LogBefore` method of `class Logger`, as pointed out by arrow ‘3’. The single object parameter of `LogBefore` is satisfied by the typed formal parameter of the `ExecJptObjParam` pointcut. A reference to this parameter is passed when `LogBefore` is invoked.



```
namespace Aspect_Logging
{
  public class Logger {

    public Logger() {}

    public void LogBefore(object param)
    {
      Console.Write("Before join point, Input:”);
      Console.WriteLine(param.ToString() );
    }
  }
}
```

Figure 3.29: Overview of advice behaviour implementation.

Advice behaviours are implemented by methods that have a parameter list corresponding to the typed formal parameters declared directly by advice and the pointcut advice references. These methods can be static, or they may be instance methods as is the case in Figure 3.29, which shows the C# implementation of `LogBefore` from the before advice in Figure 3.28. Recall that a singleton of the type implementing aspect-based property behaviour is instantiated when the woven application starts executing, which is why this type must have a public default constructor. Thus, in the full implementation of `class Logger` in Figure 3.29, the default constructor, shown in bold, is made public. The method being referenced by advice as well as its containing type should be accessible from outside the component in which they are contained. In our example, it suffices to label the method and type with a public access modifier. Typed formal parameters are passed to the method implementing advice in the order in which they are declared. That is, typed formal parameters declared by the pointcut are passed before those declared in the advice, and those of the pointcut are passed in order of declaration. To satisfy the crosscutting specification of Figure 3.28, `LogBefore` need only declare a single argument of type `Object`, but if the pointcut or advice had additional typed formal parameters a longer list of arguments would have been used.

In Figure 3.28, advice did not provide details on the result of an operation, whereas more interesting logging would include details of the results of a method execution. While the previous example could be appended with after advice, an example in which around advice is used to capture additional details will allow us to discuss the reflective and `Proceed` APIs available to around advice. In AspectJ, keywords are made available that provide access to reflective objects describing the join points and the join point context. Likewise, around advice can call upon the `proceed` keyword to pass execution control towards the join point selected for advice. In doing so, the `proceed` keyword causes the join point, or the next around advice applied to the join point, to execute. However, these keywords limit the ability to implement aspect-based properties in a variety of languages, since a language must be extended with these keywords to allow around advice and reflection can be completely supported. Our solution is to introduce an interface to support reflection and `proceed`, called `IAspect`. A specification of the interface targeted at the CLI is pictured in Figure 3.30. The reflective API mirrors that of AspectJ, with keywords mapped to correspondingly named access methods. Indeed, the same types are used to store reflective information as those used by AspectJ. It is difficult to implement the interface independent of the weaver, as the weaver provides the reflective information that determines which method is called when around advice calls `Proceed`. Also, the weaver initialises data structures that the reflective API provides access to. Thus, access to the `IAspect` is through the inheritance of an implementation provided by the weaver implementer.

```
public interface IAspect {
  org.aspectj.lang.JoinPoint              JoinPoint{get;set;}
  org.aspectj.lang.JoinPoint.StaticPart  JoinPointStaticPart{get;set;}
  org.aspectj.lang.JoinPoint.StaticPart  EnclosingJoinPointStaticPart{get;set;}

  object Proceed(params object[] obj);
}
```

Figure 3.30: `IAspect` interface that provides language-independent access to `Proceed` and reflection APIs exploited by advice.

```
<item>
  <advice>
    <around>
      <return_type>Object</return_type>
      <pointcut>
        <primitive>
          <pointcutId>
            <name>ExecJptObjParam</name>
          </pointcutId>
        </primitive>
      </pointcut>
      <behaviour>
        <name>LogAroundJpt</name>
      </behaviour>
    </around>
  </advice>
</item>
```

Figure 3.31: Around advice alternative to advice in Figure 3.28.

We discuss the methods of `IAspect` in the context of an example of an around advice crosscutting specification shown in Figure 3.31 and its advice behaviour implementation shown in Figure 3.32. Figure 3.31 clarifies the XML used to specify around advice. The advice implementation in Figure 3.32 passes control from advice to the join point midway through the execution of the advice behaviour. In this example, behaviour is written in C#, which reflects our choice of the CLI for a prototyping aspect-based properties. Control is passed when `LogAroundJpt` calls the `Proceed` method of the `IAspect` interface. An implementation of this interface is available from the `Aspect` class that the implementation of class `Logger` inherits. We should clarify that in this particular implementation, `IAspect` interface methods are not part of `class Aspect`'s public interface, and so the self reference, `this`, must be recast to an `IAspect` reference before the `Proceed` method can be accessed. This is done so that the aspect-based property writer need not worry about conflicts between method names in the aspect behaviour implementation and methods defined in the API provided by `IAspect`.

```
using TCD.CS.DSG.Weave.Reflect;

namespace Aspect_Logging
{
  public class Logger : Aspect {

    IAspect iAspect;

    public Logger() {
      iAspect = (IAspect)this;
    }

    public object LogAroundJpt(object param)
    {
      string jptDesc = iAspect.JoinPointStaticPart.toShortString();

      Console.WriteLine("Around-before join point: " + jptDesc);
      Console.WriteLine("Input: "+ param.ToString() );

      object[] args = new object[1];
      args[0] = param;

      object result = iAspect.Proceed(args);

      Console.WriteLine("Around-after join point: " + jptDesc);
      Console.WriteLine("Output: " + result.ToString());

      return result;
    }
  }
}
```

Figure 3.32: Use of runtime libraries for reflection and proceeding from around advice.

Note from its use in Figure 3.32 that the `Proceed` method has only one parameter, which is an array of object references corresponding to the arguments of the method from which `Proceed` is called. Thus, a call to `Proceed` involves obtaining object references for each of the advice method parameters and placing these references in an array of type `Object`. The `Proceed` method returns the result of the join point as an `Object` reference. The return type of the method implementing around advice must match the declared return type in the advice statement. If this is type `Object`, then the result of `Proceed` should be cast to the method's return type before it is returned. When the return type is `Void`, the result of `Proceed` is a null reference, and casting is not an issue as no result will be returned.

Figure 3.32 also includes an example use of the reflective API. In this case, a description of the join point at which advice is executing is obtained from the `JoinPointStaticPart` reference provided by the `IAspect` interface.

3.3 Summary

In this chapter, we presented a programming model for aspect-based properties. Aspect-based properties provide an AOP-based solution to tailorability and preplanning issues with contextual composition by providing a PA mechanism for writing new crosscutting functionality. Moreover, aspect-based properties can be applied to components after their design either through attributes applied prior to component compilation or via custom crosscutting specified after component compilation.

The programming model for aspect-based properties addresses reusability and language independence problems with AOP using attribute-based property selection and language-independent AOP. Aspect-based properties use attribute-based crosscutting in which pointcuts are parameterised with attributes, and attribute types are provided to access aspect-based property functionality. By specifying component-aspect bindings with attribute types, component writers avoid then the need understanding the PA mechanism underlying composition. To allow aspects and components to interoperate without regard for implementation language, the mechanism for specifying crosscutting specifications, implementing crosscutting behaviour and weaving aspects and components includes no language dependencies. Crosscuts are written in XML rather than language extensions, and pointcuts are expressed in terms of component metadata rather than language-specific types and signatures. Advice behaviour is accessed from component interfaces, and the implementation of aspect advice avoids the keywords that might introduce language-specific details. Finally, a load-time weaver is used to compose components and aspects based on details of their distribution format.

Chapter 4 Implementation

"Developers, developers, developers, developers, developers, developers, developers…"
–Steve Ballmer emphasising the importance of providing tools to support the developer.

This chapter presents the design and implementation of a prototype weaver that supports aspect-based properties for the CLI component platform. Supporting the aspect-based property programming model involves implementing a weaver for composing aspects and components in a programming environment that facilitates the implementation of programming model products such as aspect-based properties, attribute types and attributed components.

In the first section, we start by identifying elements of the CLI that support our implementation. Some products of the aspect-based property programming model correspond directly to products of the CLI's existing programming model. In particular, the CLI already provides support for implementing attributed components and attribute types. A weaver supporting a pointcut-advice (PA) mechanism is not available, but the CLI simplifies its implementation by providing some necessary infrastructure. The weaver interprets crosscutting specifications written in XML and based on the pointcut-advice mechanism described in the AspectJ V1.0.6 Programming Guide [Asp'02]. In the second section of this chapter, we explain how such an XML schema is systematically derived from a BNF specification for AspectJ V1.0.6.

Our prototype weaver, Weave.NET, addresses two problems: how to weave aspects and components and how to provide weaving at load-time. In section 4.3, we explain how we implement a weaving library that uses byte code instrumentation to weave aspect-based properties with CLI components, and inheritance to provide support for AspectJ facilities such as join point reflection and `proceed` functionality. In section 4.4, we explain how the library is integrated with the execution environment to provide a load-time weaver.

4.1 Exploiting CLI Architecture for Aspect-Based Properties

We use standardised Common Language Infrastructure (CLI) [Ecm'03b] as the target platform for our prototype weaver. The CLI already supports the ability to define attribute types, and includes support for annotating component source code with attributes among the requirements of language tools targeting the CLI. The CLI facilitates the implementation of the weaver, because the component format is suitable for clear-box crosscutting, while CLI APIs support XML processing and code generation required by the weaver.

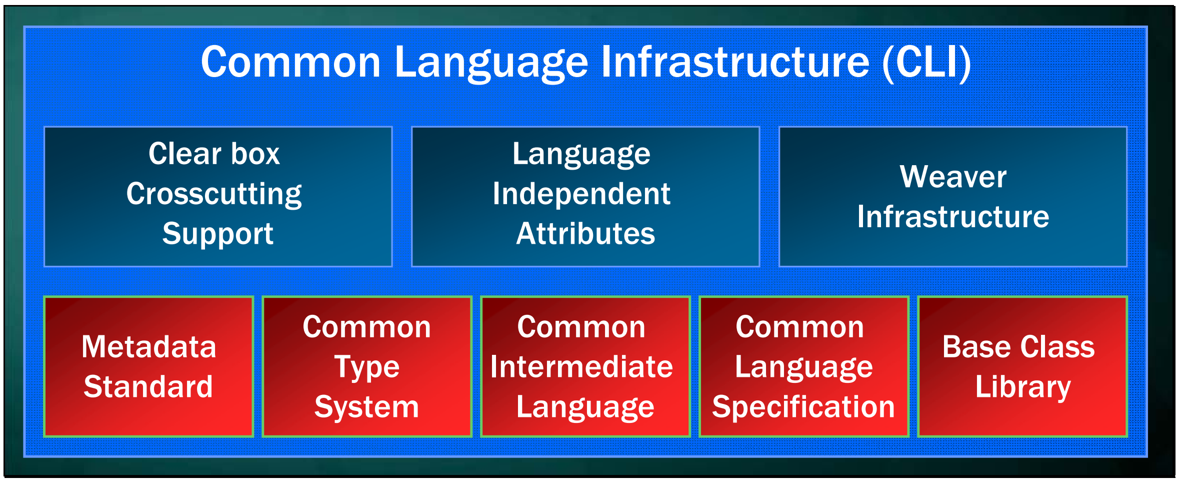

Common Language Infrastructure (CLI)
Clear box Crosscutting Support
Language Independent Attributes
Weaver Infrastructure
Metadata Standard
Common Type System
Common Intermediate Language
Common Language Specification
Base Class Library

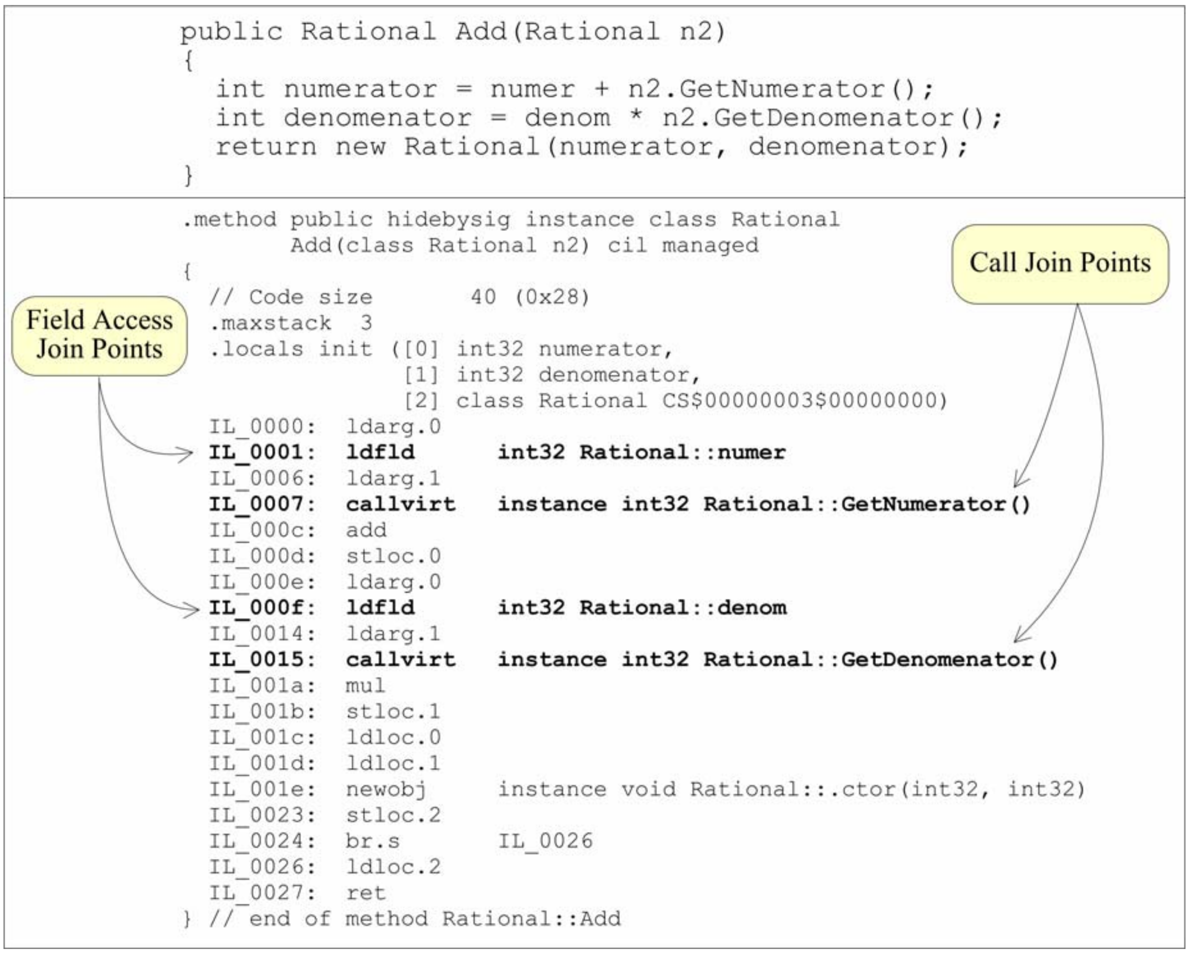
Figure 4.1: Common Language Infrastructure overview.

At the top of Figure 4.1 are the requirements met by the CLI. Clear-box crosscutting, language-independent attribute annotation and weaver infrastructure are supported in part by the CLI architectural elements identified in the bottom boxes of the figure. The internal details of CLI components suit clear-box crosscutting, as they are standardised in terms of the type system they use, the format of their metadata and the intermediate language in which method behaviour is expressed. The CLI Base Class Library (BCL) is a set of APIs provided by all CLI implementations, and among the APIs is one specifically for XML processing and code generation. Language-independent component development and annotation is made possible by the availability of a Common Language Specification (CLS) that allows the CLI to be targeted by compilers supporting a variety of programming languages. Included in the CLS are specifications that allow attributes to be defined in variety of languages and used to annotate source code of all of these languages.

Support for clear-box crosscutting allows the full range of join points available with the aspect-based property pointcut-advice mechanism to be supported. Clear-box crosscutting is supported in part by the rich metadata included in CLI components. In the CLI, components correspond to assemblies, but an assembly is only a logical unit of distribution. The actual physical unit of distribution is the module. Thus, an assembly consists of one or more modules, with each module corresponding to a single file. Assembly metadata is organised on a module by module basis with each module containing a relational database in the form of several cross referencing tables describing all structural elements including types and their members, implemented by the module [Lid'02]. The CLI's metadata specification standardises the tables describing component structure as well as the layout of these tables, which makes the metadata representation consistent for all CLI components. The expression of behaviour is also standardised in that methods are specified in terms of the CLI's byte code, called the Common Intermediate Language (CIL). More importantly, field access and method calls are emitted as single instructions that are usually parameterised with references to the metadata description of the method or field involved. These opcodes allow call and field access join points to be easily identified and distinguished according to their metadata description. As an example, let us look at the opcodes used to implement the `Add` method of a `class Rational` shown in Figure 4.2. Notice how calls and field accesses are easily distinguished by the opcodes used to specify them. The opcodes are parameterised with metadata tokens identifying the type member being manipulated; however, the disassembler that generated the bottom half of Figure 4.2 has gone a step further and replaced the metadata tokens with their text-based descriptions.

Among the Base Class Library (BCL) APIs that the CLI provides is infrastructure for parsing XML-based crosscutting specifications. This infrastructure is useful as the aspect weaver is responsible for interpreting a crosscutting specification in order to establish what modifications must be made to application components during weaving. In Chapter 3, we established XML as a suitable representation for the crosscutting semantics of aspect-based properties. Thus, a requirement of the weaver is the ability to parse the XML specification into a navigable object graph so that it may be interpreted. To simplify XML processing, the CLI implements W3C DOM navigation of XML documents [Hor'00]. In contrast to a SAX API [Sun'00], the DOM API parses XML files directly into a navigable object graph. This CLI's implementation of this API can also automatically validate the XML against an

XML Schema described in W3C XML Schema Language [Fal'01]. The grammar checking that validation provides improves usability, because it greatly reduces the amount of error checking that needs to be built into the weaver.



```
public Rational Add(Rational n2)
{
  int numerator = numer + n2.GetNumerator();
  int denomenator = denom * n2.GetDenomenator();
  return new Rational(numerator, denomenator);
}
```

```
.method public hidebysig instance class Rational
        Add(class Rational n2) cil managed
{
  // Code size       40 (0x28)
  .maxstack  3
  .locals init ([0] int32 numerator,
                [1] int32 denomenator,
                [2] class Rational CS$00000003$00000000)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      int32 Rational::numer
  IL_0006:  ldarg.1
  IL_0007:  callvirt   instance int32 Rational::GetNumerator()
  IL_000c:  add
  IL_000d:  stloc.0
  IL_000e:  ldarg.0
  IL_000f:  ldfld      int32 Rational::denom
  IL_0014:  ldarg.1
  IL_0015:  callvirt   instance int32 Rational::GetDenomenator()
  IL_001a:  mul
  IL_001b:  stloc.1
  IL_001c:  ldloc.0
  IL_001d:  ldloc.1
  IL_001e:  newobj     instance void Rational::.ctor(int32, int32)
  IL_0023:  stloc.2
  IL_0024:  br.s       IL_0026
  IL_0026:  ldloc.2
  IL_0027:  ret
} // end of method Rational::Add
```

Figure 4.2: Examples of call and field join points in CLI opcodes of a simple method.

CLI infrastructure also meets the code generation requirements of the weaver. These requirements originate from past experience with implementing weaving as inline code modifications. At that time, we noted these inline modifications proved to be a suitable means of aspect-component composition that offered good execution performance [Laf'03]. Such changes require byte code instrumentation capabilities. Rather than write a complete code generation engine, it is possible to exploit the CLI's `System.Emit` API to generate custom versions of a CLI component to suit aspect binding requirements. As we will see, the code generation library has to be complemented with a library able to introspect on existing components.

Support for attribute types comes from the ability to create user-defined custom attribute types. Attributes used to annotate types, and type elements, correspond to custom attributes in CLI terminology. Custom attributes are subtypes of `System.Attribute` that are annotated with attributes of type `System.AttributeUsage`. Annotating subtypes of `System.Attribute` with an attribute of type `System.AttributeUsage` bootstraps the type into being an attribute type, and at the same time specifies what structures it can annotate. As it corresponds to a normal type, a custom attribute has all the same interoperability properties of a CLI type. Thus, an attribute defined in one language can be used to annotate structures in another.

Support for writing attributed components is based on the CLI's Common Language Specification (CLS). The CLS lays out interoperability specifications for compilers targeting the CLI, regardless of the source code being compiled. This allows attributed components to be implemented in any programming language. Moreover, the CLS mandates that component behaviour be written in CIL, which has been designed with multi-language development in mind. For instance, instructions for tail call optimization of recursions are included to facilitate functional programming languages. As a result, there are compilers for a number of programming languages that target the CLI. Actual type-safe interoperability of attributed components is the responsibility of the Common Type System (CTS), which provides the specification to which types in the CLI are built. The CTS was developed with support for multiple programming paradigms in mind, as "the CTS supports Object-Oriented Programming (OOP) as well as functional and procedural programming languages" [Ecm'03b]. Finally, the CLI makes attributes accessible in a language-independent fashion. Using attributes for annotating definitions is different from other forms of type usage such as instantiation or member access. Annotation requires specialised language-level support from the tools being used to compile the code being annotated. Fortunately, such support is mandated by the CLS. Another minor point is that, by default, all types in the CLI have a constructor method that takes no parameters, regardless of the implementing language. This suits aspect instantiation in aspect-based properties, which relies on the presence of a parameterless constructor.

For clarity, the following subsection gives further details on how attributes are embedded into components, and how new custom attributes are defined and applied.

4.1.1 Defining Attribute Types with Custom Attributes

The CLI, with its custom attributes, provides metadata extensibility by associating arbitrary information with metadata elements. Specifically, a *custom attribute* is an instance of any subtype of `System.Attribute` that can be associated with an entry in one of the 19 CLI metadata tables, shown in Table 4.1. To control the use of custom attributes, an attribute of type `AttributeUsage` is applied to the definition of the attribute type in order to explicitly select which metadata elements instances of the custom attribute can be associated with.

Table 4.1: Metadata tables that can have custom attributes applied to their elements [Lid'02].

`Method`	`InterfaceImpl`	`Event`	`AssemblyRef`
`Field`	`MemberRef`	`StandAloneSig`	`File`
`TypeRef`	`Module`	`ModuleRef`	`ExportedType`
`TypeDef`	`DeclSecurity`	`TypeSpec`	`ManifestResource`
`Param`	`Property`	`Assembly`	

```
[AttributeUsage(AttributeTargets.Method)]
public class MakerInfo : Attribute {
  private string url;

  public string URL { get { return url; } }

  // Constructors define the positional parameters
  public MakerInfo(string url) {
    this.url = url;
  }

  // Public fields define the named parameters
  public string Programmer = "None";
}
```

Figure 4.3: Declaration of custom attribute type for storing developer details (example written in C#).

Since the definition and application of custom attributes is hard to visualise, we offer the example in Figure 4.3 of a definition of a custom attribute type in C#. The custom attribute is designed to store details about the programmer of a method. This custom attribute allows developer details such as the developer's website URL and their name to be stored in metadata. Based on the `AttributeUsage` attribute, the attribute type declared in Figure 4.3 can only be applied to methods. We should point out that the aspect-based property programming model does not make use of attribute parameters. The details in the example on how to define parameters for attribute types are for completeness, as attribute parameters will not appear in attribute types supplied by an attribute type.

The use of a `MakerInfo` attribute in a program, again written in C#, is shown in the upper section of Figure 4.4, where the application of the attribute appears in bold. Being type instances, attributes can be parameterised. In the CLI, this happens using positional parameters and named parameters [Ecm'03b]. *Positional parameters* correspond to constructor parameters used to initialise the attribute instance. So when this attribute is instantiated, `"http://www.dsg.cs.tcd.ie/~laffertd"` is used as the constructor argument. *Named parameters* correspond to field initialization values. Referring to the example, the assignment of `"Lafferty"` to the `Programmer` field corresponds to use of a named parameters.

Custom attributes are emitted to a component unchanged by the compiler. Thus, we can see the custom attribute declaration in the lower section of Figure 4.4, where the CLI equivalent of the top portion appears. Here, the attribute declaration is in bold.

```
class HelloWorld
{
  [MakerInfo("http://www.dsg.cs.tcd.ie/~laffertd",Programmer = "Lafferty")]
  static void Main(string[] args) {
    System.Console.WriteLine("Hello World!");
  }
}
```

```
method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  .custom instance void MakerInfo::.ctor(string) = (
   01 00 22 68 74 74 70 3A 2F 2F 77 77 77 2E 64 73   // .."http://www.ds
   67 2E 63 73 2E 74 63 64 2E 69 65 2F 7E 6C 61 66   // g.cs.tcd.ie/~laf
   66 65 72 74 64 01 00 53 0E 0A 50 72 6F 67 72 61   // fertd..S..Progra
   6D 6D 65 72 08 4C 61 66 66 65 72 74 79 )          // mmer.Lafferty

  // Code size       11 (0xb)
  .maxstack  1
  IL_0000:  ldstr      "Hello World!"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)f
  IL_000a:  ret
} // end of method HelloWorld::Main
```

Figure 4.4: Use of custom attribute declared in Figure 4.3 and corresponding assembly instructions.

4.2 XML Schema Design

The actual specification of crosscutting semantics is written in terms of XML. The XML schema used [Laf'02a] was developed from a BNF grammar [Est'02] extracted from the Language Semantics Appendix of the AspectJ programming guide V1.0.6 [Asp'02], and implemented in the W3C XML Schema Language [Fal'01]. As noted in Chapter 3, the AspectJ V1.0.6 grammar is supplemented to allow support for property-based crosscutting in terms of attributes. So, in addition to the schema elements derived from the BNF specification for AspectJ V1.0.6 grammar [Laf'02b], an `<attribute>` tag was added that can be chosen wherever a `signature` selection is used or a type pattern is required. This tag allows the specification of an attribute type in place of a signature or type pattern. The use of W3C XML Schema language allows more validation to occur within the XML specification than in the previous approach of using a DTD to define a schema [Vli'01]. For example, W3C XML Schema provides regular-expression syntax for restricting the values of string data, and this syntax can be used to enforce the naming rules for aspects. In contrast, the DTD data type system applies only to attributes of elements, which are fields within tags to which values can be assigned.

As much as possible, the schema used exploits the validation capabilities of W3C XML Schema. Aspects are expressed mainly in terms of XML tags rather than XML tag attributes. The organisation of the tags and their contents are defined by complex types that can then validate the grammar used to express an aspect's crosscutting semantics. Naturally, some tags such as identifiers and type patterns must contain data. These tags are described with simple types whose data is limited according to regular expressions. This removes the need to support a great deal of error checking in the weaver itself.

In the following section, we examine how conversion from AspectJ grammar to an XML schema was achieved.

4.2.1 BNF to XML Schema Conversion Rules

The conversion from BNF to XML focuses on consistency with AspectJ semantics at the expense of developing a succinct schema. In fact, conversion was done according to a systematic set of conversion rules. At the time of schema development, no systematic set of conversion rules were evident from the W3C XML Schema Language specification [Fal'01], or any related material [XML'01]. To fill this gap, we derived a set of guidelines of our own that are presented in the following sections. Note that the conversion rules assume knowledge of BNF as well as W3C XML Schema Language.

4.2.1.1 BNF Alternatives become `choice` Selections

Generally speaking, the left-hand side of a BNF production becomes an XML type containing elements defined by the right-hand side of the production. Expressions on the right-hand side that are alternatives appear in a `choice` tag. For example, the following is the BNF expression for signatures.

```
<signature> ::=        <method_signature>     |
                       <field_signature>
```

The corresponding XML type contains elements corresponding to the *<method_signature>* and *<field_signature>* set in a `choice` tag as we see below.

```
<xsd:complexType name="signature">
  <xsd:choice>
    <xsd:element name="method_signature"      type="ax:method_signature"/>
    <xsd:element name="field_signature"       type="ax:field_signature"/>
  </xsd:choice>
</xsd:complexType>
```

Notice that in W3C XML Schema it is possible to give elements the same name as their type provided that the type is not defined in the same scope.

4.2.1.2 Production Expressions become `sequence` Selections

A series of terminals and non-terminals in a single BNF expression are represented as elements grouped in a `sequence` tag. The right-hand side of the following production is the BNF expression representing before advice.

```
<before_advice> ::= [<advice_modifier>] before <formal_params>
                    ":" <pointcut> <behaviour>
```

The corresponding XML type uses the sequence tag to allow the *<advice_modifier>*, *<formal_params>*, *<pointcut>*, and *<behaviour>* tags to appear at the same time in a tag describing before advice. The XML type for *<before_advice>* is:

```
<xsd:complexType name="before_advice">
  <xsd:sequence>
    <xsd:element name="modifiers"     type="ax:advice_modifier" minOccurs="0"/>
    <xsd:element name="formal_params" type="ax:formal_params"/>
    <xsd:element name="pointcut"      type="ax:pointcut"/>
    <xsd:element name="behaviour"     type="ax:behaviour"/>
  </xsd:sequence>
</xsd:complexType>
```

4.2.1.3 Optional Items

Optional expressions, e.g., those in square brackets, are expressed by lowering the minimum number of occurrences in a type instance from the default of one to zero. An example of this is shown in the `modifier_spec` element of the XML type for field signatures presented in bold below.

```
<xsd:complexType name="field_signature">
  <xsd:sequence>
    <xsd:element name="modifier_spec" type="ax:modifier_spec" minOccurs="0"
                 maxOccurs="unbounded" />
    <xsd:element name="field_type" type="ax:type_pattern" />
    <xsd:element name="join_point_type" type="ax:type_pattern" />
    <xsd:element name="field_name" type="ax:identifier_pattern" />
  </xsd:sequence>
</xsd:complexType>
```

4.2.1.4 Keywords become Empty Tags

Terminal tokens that are keywords become empty XML elements. The XML element name then corresponds to the keyword. An alternative would be to encode the keyword in an attribute or as element data, but validity checking separate from schema validation is required to error check the data. This secondary validity checking can be simplified with the use of an enumeration, but writing the schema is still complicated as the programmer has two items to remember: the general XML element name or attribute, and the keyword data required. An additional advantage of a single XML element is that it simplifies parsing an XML specification since keywords can be identified in a graph of XML tags without the need to examine data associated with each tag node.

Based on this rule, each modifier in

```
<access_modifier> ::= public|private|protected
```

corresponds to a separate tag, as we see in the corresponding XML type:

```
<xsd:complexType name="access_modifier">
  <xsd:choice>
    <xsd:element name="public"    type="ax:empty"/>
    <xsd:element name="private"   type="ax:empty"/>
    <xsd:element name="protected" type="ax:empty"/>
  </xsd:choice>
</xsd:complexType>
```

However, we have bent this rule where there is duplication between a keyword and the containing production. For example, with advice the keyword is not required, because we know from the containing XML tags the kind of advice that is being expressed. So in the case of the before advice BNF:

```
<before_advice> ::= [<advice_modifier>] before <formal_params>
                    ":" <pointcut> <behaviour>
```

The corresponding XML sequence, shown below, does not contain an empty XML tag corresponding to the `before` keyword.

```
<xsd:complexType name="before_advice">
  <xsd:sequence>
    <xsd:element name="modifiers"     type="ax:advice_modifier" minOccurs="0"/>
    <xsd:element name="formal_params" type="ax:formal_params"/>
    <xsd:element name="pointcut"      type="ax:pointcut"/>
    <xsd:element name="behaviour"     type="ax:behaviour"/>
  </xsd:sequence>
</xsd:complexType>
```

4.2.1.5 Remove Terminal Tokens for Delimiting Text

Terminal tokens used to enclose bodies are not required, as the opening and closing tags of an XML element automatically delimit an area of text. This rule of thumb eliminates most of the round, i.e. `"("`, and curly brackets, i.e. `"{"`, from the syntax. BNF expressions that are delimited lists of other expressions are represented as a single element that can appear multiple times. Thus, the delimiters in

```
<local_refs> ::= <var_type> <var_name> { "," <local_refs> }
```

need not be represented explicitly in XML. We would represent the BNF above with an XML tag with a type containing the *<var_type>* and *<var_name>* elements, shown on the next page, and an unbounded cardinality.

```
<xsd:complexType name="local_ref">
  <xsd:sequence>
    <xsd:element name="var_type" type="ax:var_type"/>
    <xsd:element name="var_name" type="ax:identifier"/>
  </xsd:sequence>
</xsd:complexType>
```

Thus, the element can appear multiple times, which is indicated by varying the value of the `maxOccurs` attribute when `local_ref` is used as an XML tag type, as shown below.

```
<xsd:element name="local_var_ref" type="ax:local_ref" minOccurs="0"
                                              maxOccurs="unbounded"/>
```

4.2.1.6 Dealing with Informational Non-Terminal Tokens

Some BNF non-terminal tokens are included only to make the BNF more readable. Such tokens simply provide a new name for an existing BNF expression that is meaningful in the context of a particular BNF production. When converting to XML, this intermediate non-terminal token is retained by using it as the element name in the overall XML type. However, to avoid defining a new type corresponding to this element, we use the type of the expression that it maps to.

Take the BNF below in which the *<field_type>* expression is included for informational purposes only.

```
<field_introduction> ::= <intro_modifiers> <field_type>
                         <target_type> "." <identifier> [<field_init>] ";"
```

Where:

```
<field_type> ::= <var_type>
<target_type> ::= <type_pattern>
<field_init> ::= "=" <expression>
```

When converting to XML, a new XML type is not defined for the `field_type` non-terminal. Instead, an element of type `var_type` is used as shown below.

```
<xsd:element name="field_type" type="ax:var_type "/>
```

4.2.1.7 Unary and Binary Operators are Tags

Unary and binary operators, exemplified by the logical operators employed by AspectJ, are expressed using attributes and elements respectively. Thus, the logical `not`, i.e. `"!"`, is represented as an attribute of an XML type to which it can be applied. The attribute is a `Boolean` value with a default of `false`, which avoids any need to validate the attribute's data. In contrast, the logical *or* and logical *and* operations, i.e. `"||"` and `"&&"` in AspectJ syntax, are expressed with an XML tag type with the corresponding name. The XML elements are defined with anonymous types containing the operands of the logical function. For example, the logical 'or' available in pointcut expressions, shown below

```
<pointcut> ::=        <primitive_pointcut>          |
                      "(" <pointcut> ")"            |
                      "!" <pointcut>                |
                      <pointcut> "&&" <pointcut>    |
                      <pointcut> "||" <pointcut>
```

corresponds to the following XML type:

```
<xsd:element name="or">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="pointcut" type="ax:pointcut" minOccurs="2" maxOccurs="2"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

4.2.2 XML Summary

The XML rules discussed in the previous section have been applied to creating an XML schema. As mentioned, this schema is listed in Appendix A of this thesis.s

4.3 Weaving Library

In this section, we focus on the weaving library underlying Weave.NET. This library must provide an API that generates a property-bound component from an aspect-based property and a component, be it an attributed component or a legacy component. Recall that property-bound components are components that have been inspected at load-time for join points matching the pointcut specifications of aspect-based properties. Matching join points are bound to aspect-based property behaviour according to advice. In the case of the CLI, components are loaded when one of the types they implement is required by the execution environment. The weaving library must also provide an implementation of the `IAspect` interface. Recall that in Chapter 3, we pointed out that the `IAspect` implementation requires cooperation on the part of the weaver to guarantee that reflective data structures are properly initialised to allow introspection on join point state and to allow execution control flow to be influenced by `Proceed` calls in around advice.

In the following sections we present the weaving library's interface, and explain how the interfaces are implemented. In section 4.3.1, the library APIs are presented. In section 4.3.2, an overview is presented of the architecture that allows the weaver to match join points with pointcuts and apply relevant advice. However, not all matching decisions can be made at weave time, and so the execution of advice is not always a static decision. In 4.3.3, we examine the algorithms that are embedded into the woven component to allow runtime decision making associated with selecting join points for `cflow` pointcuts and executing around advice. This section also includes details on how the weaver provides runtime support for the `IAspect` interface. Finally, section 4.3.4 points out limits to the weaver due to its single pass code generation design, its use of static pointcut matching and limits to developer resources available during its implementation.

4.3.1 Weaving Library APIs

The weaving library provides two APIs. A weaving API implemented by the type `TCD.CS.DSG.WeaveDotNet` generates woven assemblies, and an `IAspect` implementation by the type `TCD.CS.DSG.Weave.Reflect` supports reflection and `Proceed` functionality for around advice.

4.3.1.1 `TCD.CS.DSG.WeaveDotNet`

The API of the prototype weaving library is shown in Figure 4.5. The library is stored in a CLI assembly composed of a single .DLL file called `wdn.dll`. This API accepts as input a reference to a CLI assembly and to an XML document that contains the specification for a particular aspect. The reference to the CLI assembly corresponds to a full filename of the assembly's primary module, which includes the extension, along with the path to the file relative to the current directory of the execution environment. The XML file is referenced in the same way. By default, the current directory of the execution environment is the same as the current directory when the CLI execution environment was launched. The Boolean value returned by the method indicates whether weaving failed or succeeded. More importantly, a result of true means that a new version of the component assembly has been created. The new assembly is placed in the current directory.

```
namespace TCD.CS.DSG {
  public class WeaveDotNet {

    public static bool Weave(string targetFile,
                             string targetPath,
                             string xmlFile,
                             string xmlPath) {
      ...
    }
  }
}
```

Figure 4.5: Weaving interface for prototype weaving library.

Like BCA [Kel'98], our weaver generates components that are binary compatible with their original version. This allows newly woven components to work properly with client components that have already been loaded. Without binary compatibility, the modularity of a component will be broken, as components will no longer link properly with the rest of the application based on their API. So, binary compatibility avoids the need to synchronize clients of the component being woven with the changes made.

4.3.1.2 **`TCD.CS.DSG.Weave.Reflect.IAspect`**

The `TCD.CS.DSG.Weave.Reflect.IAspect` interface implemented by the prototype weaver is shown in Figure 4.6. This interface is a forerunner to the standard version of the `IAspect` interface described in Chapter 3, as it includes deprecated helper functions from earlier versions of the prototype weaver. These helper methods provide access to join point metadata for the implementation of `Proceed`. Using helper functions, the metadata can be obtained directly from an `IAspect` reference as opposed to a reference to its implementing type. This allows the weaver to avoid emitting recasting instructions during code generation, which was of interest when the weaver was initially implemented. With a small amount of programming effort these helper functions could be removed from the weaver, but they have been left in place as they do not limit prototype functionality.

```
public interface TCD.CS.DSG.Weave.Reflect.IAspect {
  org.aspectj.lang.JoinPoint             JoinPoint{get;set;}
  org.aspectj.lang.JoinPoint.StaticPart  JoinPointStaticPart{get;set;}
  org.aspectj.lang.JoinPoint.StaticPart  EnclosingJoinPointStaticPart{get;set;}

  object Proceed(params object[] obj);

  // Control structures specific to Weave.NET implementation of proceed.
  void QueueAroundAdvice( int[]                 mappings,
                          RuntimeMethodHandle   adviceMeth,
                          RuntimeFieldHandle[]  cflowActiveFlagRefs,
                          bool[]                cflowActiveFlagStates);

  bool       ExecJoinPointRecurseActive { get;set; }
  object[]   ProceedParams{ set;}
  RuntimeMethodHandle JoinPointInvocationCache{set;}
}
```

Figure 4.6: Prototype weaver's specification for the `IAspect` interface.

This `IAspect` interface is implemented by the class `TCD.CS.DSG.Weave.Reflect.Aspect`, which is available from a CLI component contained in a single file called `wdnr.dll`. This class is discussed further in 4.3.3.3.

4.3.2 Weaver Architecture

Architecturally, the weaver decomposes into two principle systems: one for code generation and the other for aspect modelling. The code generation system performs a traversal of a software component targeted for weaving in which the existing implementation is copied into a new version of the software component. During the traversal, join points and potential join points are identified by the aspect modelling system, which is responsible for interpreting the XML-based crosscutting specification. Potential join points require a runtime decision to determine whether or not to execute crosscutting functionality, whereas known join points can be directly bound to aspect behaviour. These modifications are carried out under the direction of the aspect modelling system by the code generator.

The bridge between these two systems is the `JoinPoint` class hierarchy. The code generation system creates instances of objects in this hierarchy to encapsulate join point details, which are passed to the aspect modelling system for examination. Join point objects also provide code generation capabilities specific to join points for embedding advice that used is by the code generation system.

In section 4.3.2.1, we will review the code generation system and explain how it interacts with `JoinPoint` objects. In section 4.3.2.2, we will review the aspect modelling system, and examine how it interacts with `JoinPoint` objects to decide whether or not to apply advice. The `JoinPoint` objects themselves are discussed in section 4.3.2.1.1, where we summarise the purpose of each type in the `JoinPoint` type hierarchy.

4.3.2.1 Code Generation Architecture

The code generation system performs a traversal of a software component targeted for weaving in which the existing implementation is copied into a new version of the software component. The traversal is at the level of CIL (byte code), meaning that methods are copied one instruction at a time rather than in blocks of instructions. Traversing component behaviour at the CIL level allows intra-method join points to be exposed to pointcuts for the purpose of clear-box crosscutting. This traversal requires that the code generation system have a means of examining a component without loading it into the execution environment, since we will want to load the newly generated component instead. Also, there should be a means of generating a new component that will be the result of weaving.

The new component is created using a CLI dynamic assembly, which is one whose implementation can be specified at runtime. The CLI's `System.Reflection.Emit` API [Ecm'03b] provides classes to create an object graph corresponding to a dynamic assembly. The principle classes used by the `Emit` API to model a dynamic assembly are shown in Figure 4.7. Here, a module corresponds to a physical file. Thus, an assembly can span files. Types and their constituent members are contained entirely within one module. Were it not for the modifications specified by the aspect, a hierarchy of these objects built from the component being woven could be emitted without change; however, as per the aspect, there will be some differences.

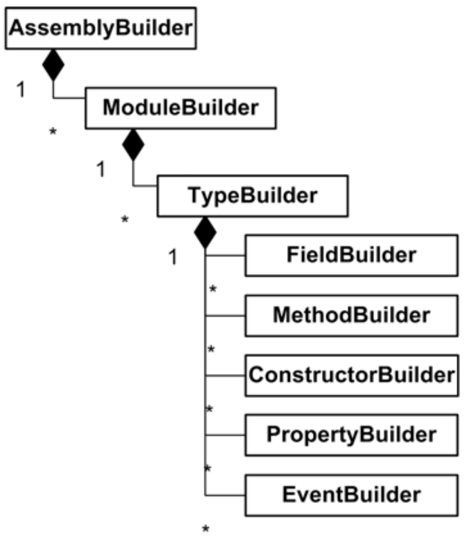


Figure 4.7: Dynamic assembly as modelled by the `System.Reflection.Emit` library.

To inspect the existing assembly, a third-party library called the CLIFile Reader API [Cis'02] was used. The `System.Reflection` API has been suggested as a tool for introspecting on existing assemblies [Sch'02], but this API lacks the ability to directly access the CIL stream. Without access to CIL it is impossible to expose call, field access and object pre-initialization join points, which are expressed as method invocations or field accesses in the body of methods. Thus, the code generation system bypasses the convenience of the `Reflection` API and examines the assembly metadata directly using the CLIFile Reader API. The CLIFile Reader API provides abstractions to access intra-method details such as the CIL stream and the method's exception handling table directly from the CLI file format [Lid'02]. The benefit of using the CLIFile Reader over directly accessing the file from the weaver is that the CLIFile Reader provides decompression and metadata table modelling, and it greatly simplifies resolving cross-references within table entries.

The code generation system must map assembly details provided by the CLI File Reader to their representation in the `Emit` API, but there is a mismatch in the way the two APIs model assemblies. The object graph used to model an assembly in the CLIFile Reader corresponds to the organisation of metadata. Since metadata is organised on a module basis, type members are keyed with module-wide identifiers that do not immediately identify their containing type. In contrast, the `Emit` API expects a type to directly reference its constituents. To bridge these two views, the code generation system introduces its own object graph for modelling an assembly based on the class hierarchy

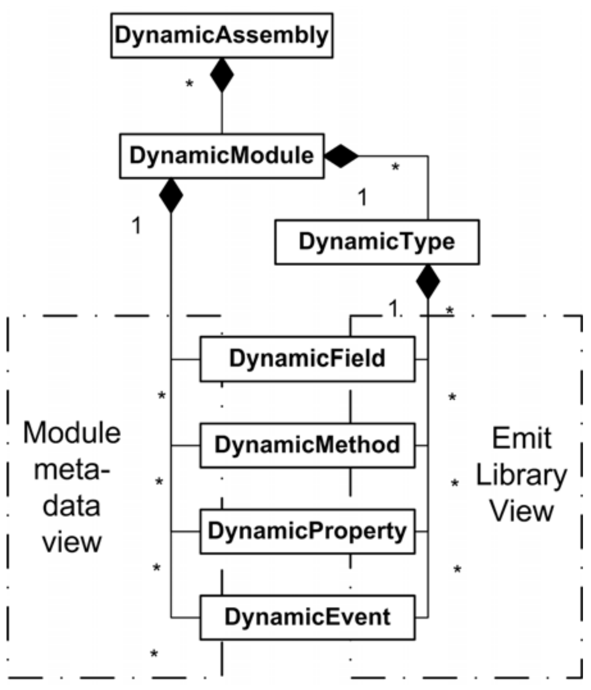


Figure 4.8: Architecture to resolve `Emit` object hierarchy and CLI metadata indexing.

shown in Figure 4.8. This hierarchy mirrors the `Emit` API. Indeed, each class in Figure 4.8 aggregates the corresponding `Emit` API object. For example, instances of `DynamicType` contain an instance of `System.Reflection.Emit.TypeBuilder`. These aggregate relationships allow instances of the classes in Figure 4.8 to be used for code generation. On the other hand, objects of the class hierarchy in Figure 4.8 retain the relationships necessary to look up objects using metadata keys. For instance, objects of type `DynamicModule` have arrays that allow lookup of `DynamicType` instances as well as `DynamicMethod` instances using the module-wide metadata keys used in the original assembly that is being woven. The availability of metadata key lookup is required to emit CIL instructions that have a metadata token as their parameter. This is because the `Emit` API provides special methods for emitting token-dependent opcodes that take the corresponding builder object as a parameter. Doing so accounts for the fact that the newly generated assembly may key its metadata differently based on whether the types being implemented are changed slightly. An example of this approach to emitting opcodes can be seen for method invocations. Method invocations in the existing file are expressed as a call opcode followed by a metadata tag corresponding to the method being invoked, whereas in the code generation API an invocation corresponds to an opcode and a reference to the `MethodBuilder` object of the method being called.

During CIL traversal, the code generation system detects join point implementations, sometimes referred to as join point shadows [Hil'04], and models them with objects of type

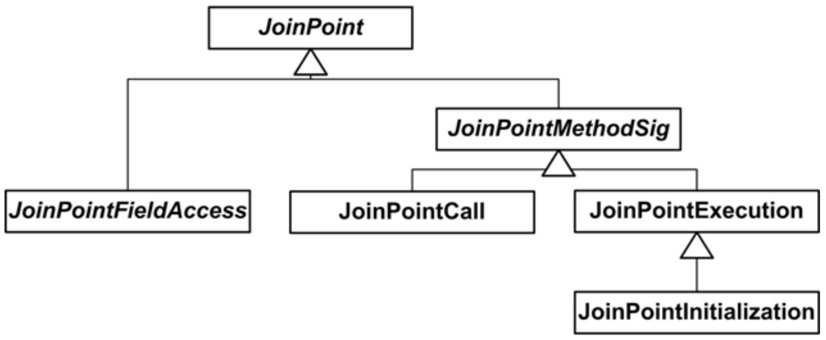


Figure 4.9: `JoinPoint` class hierarchy.

`JoinPoint`. The join points are modelled by the class hierarchy defined in Figure 4.9, where `JoinPoint` and `JoinPointMethodSig` are abstract classes. Objects of this hierarchy allow pointcut matching and code generation for advice invocations. The objects in the `JoinPoint` class hierarchy are well understood by the aspect modelling system, which provides an API that compares the objects to pointcuts in an aspect and loads the join point objects with details that allow matching advice to be called. The `JoinPoint` class hierarchy objects also have an API that generates the code required to invoke advice associated with the `JoinPoint`. This code marshals advice parameters and calls the method that implements aspect advice. The code generation system accesses this API before and after it emits the code corresponding to the join point.

In the following subsection, we characterise the objects of the Figure 4.9 class hierarchy in terms of CIL.

4.3.2.1.1 Join Point Types Expressed in CIL

In Figure 4.9 four concrete classes are used to model the join points that appear when CIL is traversed. The classes are `JoinPointExecution`, `JoinPointCall`, `JoinPointFieldAccess`, `JoinPointInitialization`. Objects of these classes model different join point types depending on the CIL instructions at the point that the object is instantiated.

`JoinPointExecution` objects correspond to blocks of CIL. In a .NET assembly, CIL code is located on a method by method basis. The assembly's metadata identifies which block of CIL code corresponds to which method signature. This is true for constructors as well, since constructor bodies are modelled as methods with special names, such as `.ctor` in the case of an instance constructor, and by marking the methods with certain metadata flags that distinguish them from other methods. Fine-grained execution join points are resolved by closer inspection of the implementation of the method body. In the case of exception handlers, extra metadata tables associated with the method’s opcodes identify blocks of exception handling code. For execution join points related to object instantiation, it is necessary to examine the CIL at the start of the constructor to distinguish constructor execution from object initialization and initializer execution join points. This is because data member initialization and flow of control between different constructors in a class’s inheritance hierarchy are written explicitly into each constructor method.

`JoinPointCall` objects correspond to join points that are present on the calling side of a method invocation or when the `new` operator is called for object construction. These points are observed as CIL opcodes of type `InlineMethod`, which indicate the target method with a metadata token. Using this token, it is possible to look up the signature of the method being called. The signature also indicates where on the stack the call context is located. Constructors present a special case. They may be accessed as part of a call join point, for instance as part of a `new` operation, or they can be accessed as part of an execution join point, for instance via `this()` and `super()` calls in Java. Fortunately, these two cases are distinguished by the opcode used to access the constructor, which is `NewObj` in the case of a constructor call join point. Object pre-initialization join points also present a special case, as they are method invocations whose parameters are used in calls to other constructors, such as `this()` and `super()`in the case of Java [Lad'03]. Essentially, the code generator must be aware of whether or not the result of a call is going to be used as a parameter for calls to other constructors.

`JoinPointFieldAccess` objects correspond to a read or write access to a data member, or field in CLI terminology. These join points do not include `final` fields, i.e., constant fields emitted as literals in CIL. These join points are observed as special CIL opcodes used to access static and non-static fields. These opcodes are associated with a metadata token identifying the signature of the field being accessed.

The `JoinPointInitialization` specialisation is of particular interest. It exists because code generation for advice applied to object initialization join points varies slightly from that applied to execution join points. In the context of object construction, execution join points correspond to the execution of a constructor body or data member initialization, whereas object initialization join points correspond to the execution of all the constructors used to create an object. Thus, object initialization join points include the execution of constructors defined in the object’s class as well as those defined in inherited classes. The difficulty is in knowing where the initialization join point ends, since this is not associated with completion of a particular constructor but rather a set of constructors. Take for example a class with two public constructors `ctorA` and `ctorB`. Typically, constructors all have the same name, but for illustrative purposes we make an exception. In this example, `ctorA` calls `ctorB` and `ctorB` calls an inherited constructor. In this case, there are two different, overlapping implementations of object initialization join points. One starts and ends in `ctorA`, and the other starts and ends in `ctorB`. Assume we apply the same advice to object initialization join points that begin with a call to `ctorA` as well as object initialization join points that begin with a call to `ctorB`. When execution reaches the end of `ctorB`, advice should only be executed if execution does not return to `ctorA`. Rather than examining the stack to see if execution will return to `ctorA`, code generation implemented by `JoinPointInitialization` makes use of a reference counter that is added as a field to the type containing an initialization join point. Essentially, the counter tracks the depth of the constructor call graph during object instance initialization. The implementation of this counter is naïve in that the counter is always created for a type. Note that the counter must also be made thread safe, but thread safety is easily implemented in the CLI by annotating the field with a thread static storage attribute, which is understood by the execution environment.

4.3.2.2 Aspect Modelling Architecture

The aspect modelling architecture is responsible for matching join points to advice according to XML-based crosscutting specifications. The aspect modelling system puts the crosscutting specifications into an object graph that makes it simple to traverse all the pointcuts for the purposes of matching, to determine the implementation advice corresponding to the matching pointcuts, and to match up join point context variables with parameters required to invoke matching advice. The relationships between classes used to model crosscutting specifications and store advice/join point bindings are shown in the UML class diagram of Figure 4.10.

The top portion of Figure 4.10, labelled ‘1’, is generated directly from the aspect’s XML-based crosscutting specification. The CLI provides the `System.Xml` library for modelling XML documents and `System.Xml.Schema` for modelling XML Schemas specifically. This XML API parses an XML file into an object graph. The object graph is built in accordance to the W3C DOM [Hor'00] standard for navigating an XML document. During XML file parsing, the XML is validated. References to nodes in the object graph generated are stored in the `Aspect`, `Advice`, `NamedPointcut` and `TypedFormalParam` objects in Figure 4.10.

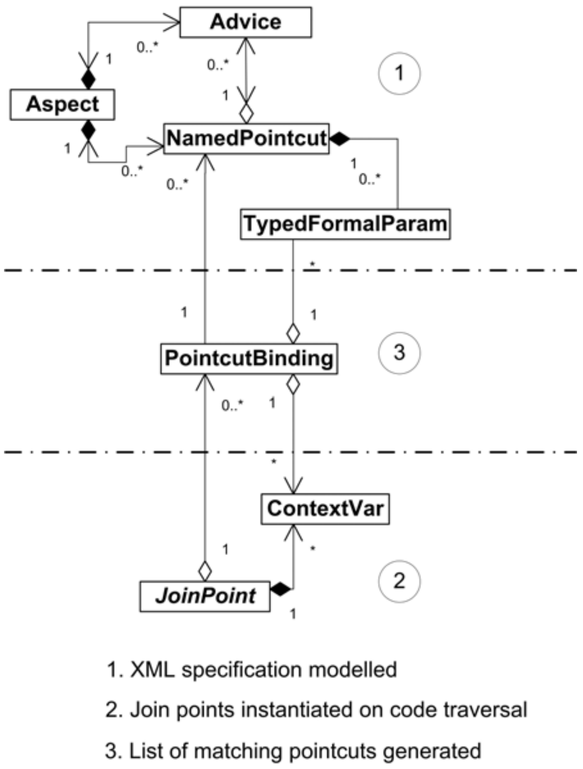

Advice
Aspect
NamedPointcut
TypedFormalParam
PointcutBinding
ContextVar
JoinPoint
0..*
1
*
1
2
3
1. XML specification modelled
2. Join points instantiated on code traversal
3. List of matching pointcuts generated

Figure 4.10: Aspect modelling and join point matching architecture.

Let us elaborate further on the `Aspect`, `Advice`, `NamedPointcut` and `TypedFormalParam` classes in Figure 4.10, starting with the `Aspect` class. An `Aspect` object stores details that are the same for all advice and named pointcuts such as a reference to the type implementing the advice behaviour. This object provides facilities to the code generation system for aspect instantiation, which involves defining a new type with a single field that is statically initialized with an instance of the aspect type. Here, instantiation is performed via a static constructor for the type, as the CLI guarantees that static constructors are instantiated any time before the associated type is referenced. Specialisations of the `Advice` class exist for each kind of advice. Having the ability to modify existing assemblies allows all types of advice to be supported. Specific details on code generation for advice invocations are covered in the next section. Recall that all pointcuts in our programming model are named pointcuts. These are modelled with `NamedPointcut` objects. These objects retain a reference to the XML element corresponding to the root of the pointcut description, and they also reference `TypedFormalParam` objects describing the pointcut's typed formal parameters. `TypedFormalParam` objects describe the context variables that the pointcut exposes to advice.

The bottom portion of Figure 4.10 is instantiated by the code generation system each time a join point is identified. The `JoinPoint` object contains the join point's signature and references to `ContextVar` objects describing the variables in the context of the join point. These variables are the parameters used to activate the join point as opposed to the set of all accessible variables within the scope of the join point. For example, for a method the `ContextVar` objects corresponds to method parameters and not variables declared in the scope of the method body.

The relationship between a join point and its advice is stored in a `PointcutBinding` object, shown in the centre of Figure 4.10. This object is generated when the `Aspect` object traverses its list of named pointcuts asking each to determine if the join point is selected by its pointcut declaration. Join points are matched with named pointcuts, which in turn maintain references to `Advice` objects that reference the named pointcut. References to these `Advice` objects are added to the `PointcutBinding` object, along with a mapping between the join point context variables and the typed formal parameters required to call advice, which must match the join point in type and number. The prototype supports the full range of typed formal parameters that can be exposed using `args`, `this` and `target` primitive pointcut designators.

4.3.3 Runtime Support

Runtime support corresponds to instructions that the weaver adds to the component being woven. These instructions are responsible for invoking advice, implementing runtime support for `cflow` pointcuts, and implementing the `IAspect` interface supported by the weaving library. In this section we give an overview of how this runtime behaviour works.

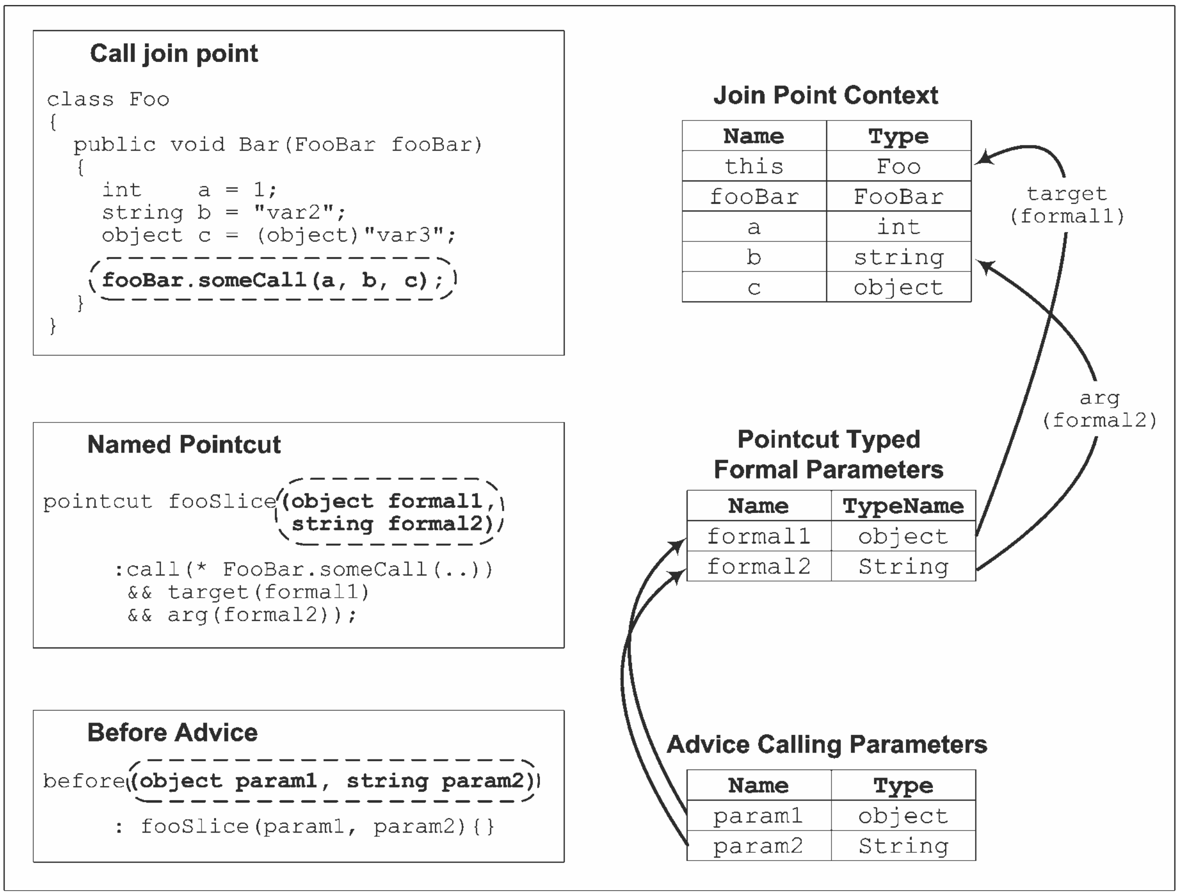
4.3.3.1 Binding Advice to Join Points

The weaving library implements advice by transferring control to the method implementing advice behaviour, rather than copying the advice's implementation to the join point shadow. This is done primarily to simplify the implementation of weaving. Copying advice CIL into methods at join point shadows involves guaranteeing access to fields and methods in the aspect type referenced by the advice. Field references would have to be rewritten as accesses to fields in the aspect instance, which might also involve changing the access privileges of aspect type members where aspect type fields are private. The same is true for methods called by the advice, so where these methods reside in the aspect type the access privilege might have to be changed. Furthermore, advice code would have to be examined for exception handling blocks, as these are emitted with special code generation functions that update the exception handling tables of methods into which the advice's implementation is being copied. Fortunately, transferring control to the method implementing advice avoids these complications.

In the following subsection, we examine how details required to call advice are obtained. Next, we look at the special difficulties associated with implementing around advice.

4.3.3.1.1 Determining Advice Invocation Details

Transferring control to advice involves emitting a call to the method implementing advice. This method's name, containing type, and implementing assembly are explicitly specified in the crosscutting specification. Advice can be supported by either static or instance methods, as the advice method metadata indicates whether an aspect instance is required in order to call the advice method or not. For instance methods, the aspect singleton is used to invoke the method. This singleton is referenced from a global type that the code generation system built using the `Aspect` object in the aspect modelling system.



```
class Foo
{
  public void Bar(FooBar fooBar)
  {
    int    a = 1;
    string b = "var2";
    object c = (object)"var3";
    fooBar.someCall(a, b, c);
  }
}
```

```
pointcut fooSlice(object formal1,
                  string formal2)
     :call(* FooBar.someCall(..))
      && target(formal1)
      && arg(formal2));
```

```
before(object param1, string param2)
     : fooSlice(param1, param2){}
```

Name	Type
this	Foo
fooBar	FooBar
a	int
b	string
c	object

Name	TypeName
formal1	object
formal2	String

Name	Type
param1	object
param2	String

Figure 4.11: Back tracing advice parameters to join point context variables (crosscutting specifications written in AspectJ-like syntax for clarity).

Before invoking the advice method, references to the parameters for invoking advice must be placed on the stack. The first step to generating code that places these parameters on the stack is to determine which `ContextVar` objects the parameters correspond to. This involves backtracking from the advice method parameters to the pointcut specification referenced by advice and then to the join point context variables that match the pointcut. The backtracking process is visualised in Figure 3.11. Note that the advice and pointcut declarations in the figure are written in AspectJ syntax for clarity.

Typed formal parameters and advice method parameters map one to one according to their declaration order, but matching typed formal parameters to join point context variables requires more work as there may be more context variables than typed formal parameters. Typed formal parameters, modelled by `TypedFormalParam` objects, are assigned variables in the context of a join point via the `args`, `this`, or `target` contextual primitive pointcut designators. For example, `this` is the self reference variable in the context of the join point, and may or may not exist depending on whether the join point is executing in the context of an object instance. `target` is the variable used to reference a method or field being accessed. As with `this`, a `target` variable may or may not exist. Note that for certain join points such as execution join points, `this` and `target` may reference the same execution context variable. `args` implies that the arguments of the join point are being bound to a typed formal parameter.

Since the meaning of these contextual primitive pointcut designators varies according to the join point type, the join point object decides what variables in the execution context are available. These context variables are modelled with `ContextVar` objects. Which `ContextVar` objects match a primitive pointcut designator is determined by the `JoinPoint` object and not the `ContextVar` object itself, and this information is used at weave time to generate the mapping between the `TypedFormalParam` objects of a pointcut and the `ContextVar` objects of a join point. The mapping is then cached in the `PointcutBinding` object.

In terms of code generation, the `ContextVar` object is responsible for generating code to place a reference to a context variable on the call stack, while the `JoinPoint` object is responsible for generating code to cache all context variables. Depending on the join point type, context variables may be on the stack or they may be an argument of the method executing. Also, a context variable reference may have to be pushed onto the stack multiple times, as any given join point may be influence by multiple pieces of advice. For these reasons, it is best if the each concrete specialisation of the `JoinPoint` class specialisation provides functionality to generate code that caches all context parameters. By using the cache as the source for context variable references, `ContextVar` objects can use the same implementation for accessing context variable references, regardless of join point type.

The weaving library's implementation of typed formal parameter to context variable mapping had to make subjective decisions to resolve ambiguities in the AspectJ Programming Guide [Asp'02]. The programming guide is unclear as to whether typed formal parameters can be used in primitive pointcut designators other the contextual primitive pointcut designators, i.e. `args`, `this`, and `target`. Nor are typed formal parameters explicitly restricted to only appearing once in a pointcut. For instance, if two primitive pointcuts are combined with a logical *or*, it might be tempting to let the same typed formal parameter be used as an argument for both primitive pointcuts and assume that the variable is bound to according to the first match. We assume only a contextual primitive pointcut designator can take typed formal parameters as their arguments and that a typed formal parameter can only appear once.

4.3.3.1.2 Exposing Join Points for Invocation by **Around** *Advice*

When around advice executes, it has the option of passing control to the next piece of applicable around advice, or where no more around advice remains to the join point itself. Around advice exercises this option by invoking the `Proceed` method of the `IAspect` interface, whose arguments correspond to the join point context used by the advice calling `Proceed`. Activation in this manner allows the join point to return to the method implementing the around advice. This situation is shown in Figure 4.12, where the execution flow is visualised for all three categories of advice.

As evident from Figure 4.12, there is a requirement that the join point somehow be exposed as a separate method that does not include around advice so that it can be invoked by the around advice without the risk of infinite recursion. The weaving library meets this requirement by defining a method that corresponds to the join point. The best option is to create this method as a static method in the component being woven. Changes to the aspect type are ruled out by our decision to use aspect types 'as is', and an instance method in the component is problematic where the join point is implemented by code in a static method.

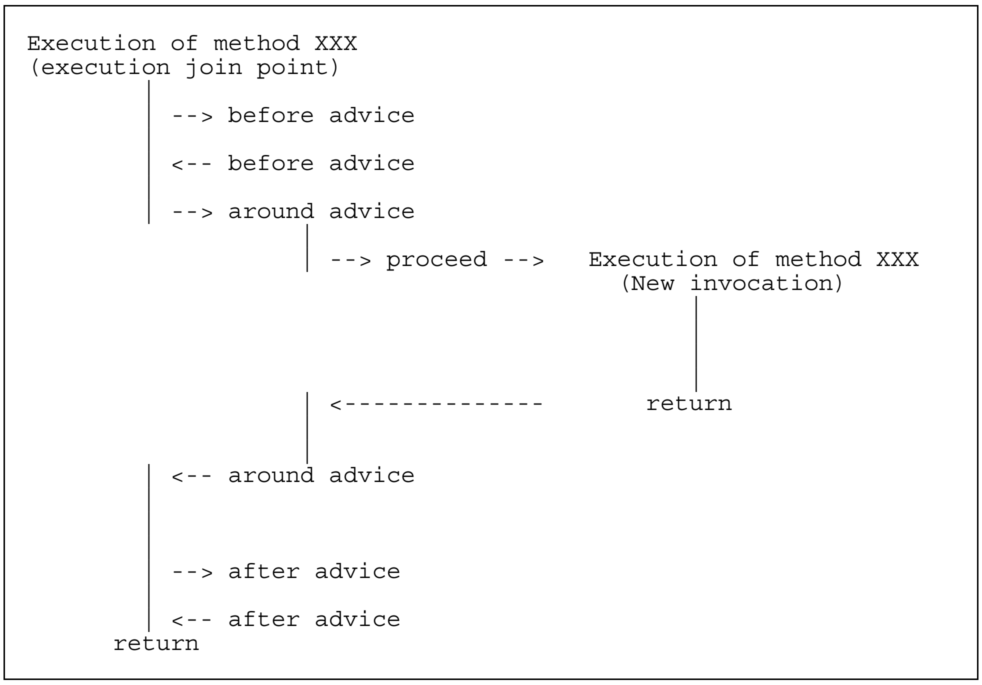


Figure 4.12: Visualisation of invocation of different categories of advice for an execution join point.

In our implementation, we refer to this method as the join point's *direct invocation method*, and the direct invocation method's body performs three basic tasks: marshalling of parameters required to execute the join point, execution of join point behaviour, and marshalling of results. The marshalling of parameters is generic across all join point types, and consists of pushing all join point execution parameters onto the stack. A shortcut for code generation is to make these parameters those of the direct invocation method.

The means by which the direct invocation method implements join point behaviour varies according to the join point type, with call and field access join points being straightforward to implement. With call join points, implementation of the join point exposure is a matter of knowing the method being referenced in the call join point. This reference identifies if the method is static or not, which makes it easy to select the proper CIL instruction for invocation. With field access join points, the implementation requires knowledge of whether the join point is a get or a set. This can be ascertained from the `JoinPoint` object, which also can provide the metadata token operand for the CIL instruction used to access the field.

Recreating an execution join point's implementation in the direct invocation method is difficult due to the complexity of execution join points, so the direct invocation method calls the original method containing the execution join point. Doing so requires care to avoid executing unwanted code and to avoid infinite recursion on around advice embedded in the original execution join point. One solution considered for avoiding recursion was to modify the execution environment to add a CIL opcode that allowed an invocation to return without destroying its stack state. A second instruction would also be added to allow the method to be re-entered. Using such instructions we could call around advice from the start of an execution join point, then temporarily return to complete executing the join point before finally resuming around advice after the join point completes. This solution was avoided as it would make the weaver platform specific. Instead, the existing join point implementation is adjusted to bypass advice when around advice causes recursion. Reference counting logic is added to ensure that the around advice is not executed when `Proceed` is called. Before and after advice is shut off as well to ensure that they are not doubly executed.

Conditional branch instructions placed in the body of the execution join point detect whether the method is being called by a direct invocation method or as part of normal application control flow by examining a recursion bypass flag. This flag is held in a new field defined in the type containing the join point's implementation. This flag is set by the direct invocation method before calling the method implementing an execution join point. When the flag is set, around advice is bypassed and the flag is immediately reset so that around advice will be properly executed during subsequent recursive calls. Because this automatic reset destroys the flag's original value, the flag has to be cached in the context of the execution join point if it has to be inspected again, for instance to decide whether or not to execute after advice.

Marshalling the return parameters is trivial if the return type of the direct invocation method is the same as the join point. If this is the case, then after the direct invocation method completes, its result will be left on the stack.

4.3.3.2 Implementing `cflow` Semantics

Matching between join points and `cflow` primitive pointcut designator specifications has the property that matching is dependent on runtime information, and so it is often the case that only potential matches can be determined during the load-time transformation. In this section we clarify the semantics of a `cflow` primitive pointcut designator and then discuss algorithms to find potential matches and determine whether they match at runtime.

4.3.3.2.1 Understanding `cflow`

`cflow` is an extremely powerful primitive pointcut designator in that it exposes join points implicitly as well as explicitly. Those join points that match the pointcut argument of a `cflow` are considered explicit matches, while join points that are traversed in the course of executing any of the explicit join points are considered to be implicit matches. The set of implicit join points is unrestricted, so any type of join point is selected by `cflow` be they a in the category of field access, execution or call.

It is easy mistake the `cflow` pointcut specification as a solution for applying advice to the beginning of a recursive call tree. This mistake is caused by assuming that `cflow` has similar properties to the `initialization` primitive pointcut designator, which begins at the first invocation of a constructor during object initialization and ends when that constructor returns. Under this assumption, `cflow` advice would only be executed before and after the first method invocation in a recursive call chain. In fact, `cflow` advice is executed for each recursive call.

In contrast to the AspectJ V1.0.6 implementation, our prototype weaver does not consider advice execution when matching join points. If advice were examined for pointcut matches, an unrestricted `cflow` primitive pointcut designator would attempt to apply advice to the execution of advice, which would cause an infinite loop. An example of such a situation appears below in the `Cflow` aspect, which is written in AspectJ syntax. With AspectJ V1.0.6, the `AddCflow` pointcut is itself executed in the context of the `AddCflow` join points. Thus, application of the advice will result in an infinite recursion on the `before` advice.

```
public aspect Cflow {
  pointcut AddCflow():  cflow( execution( * Foo.FooBar( * )));

  before(): AddCflow() {
      System.out.println("Before advice,"
                  +thisJoinPointStaticPart.toLongString() );
   }
}
```

As the language specification is ambiguous about whether advice is examined for join points, this restriction does not does not constitute a limit to our weaver.

`cflowbelow` is another primitive pointcut designator that selects join points according to whether they appear in the execution flow triggered by another join point. `cflowbelow` differs from `cflow` in that join points explicitly matching the `cflowbelow` designator's argument are not included in the set of join points to which advice is applied. Our implementation of `cflowbelow` is consistent with that of AspectJ V1.0.6 as neither considers advice execution when matching pointcuts.

4.3.3.2.2 Design Considerations for `cflow` Matching

The difficulty with weaving pointcuts specified with a `cflow` or a `cflowbelow` designator is establishing which join points implicitly match the pointcut. In this section, we concentrate on algorithms to detect matches with the `cflow` designator as doing so for the `cflowbelow` designator is a subset of the same problem.

At weave time, we are interested in identifying join points that match the `cflow` designator's argument explicitly, and those join points on which explicitly matching join points are in turn dependent upon for their proper execution. Explicit matches to a `cflow` designator, referred to as *explicitly matching join points*, can be made by comparing the join point's signature and its immediate context directly with the `cflow` designator's pointcut argument. Since explicit matches are based on the static properties of a join point, the match can be done prior to execution.

For implicitly selected join points, referred to as *implicitly matching join points*, the metadata describing the join point's implementation does not determine a match to a `cflow` designator specification, since these details do not indicate if the join point is executing as part of an explicitly matching join point or an implicitly matching join point on which an explicitly matching join point is dependent. It is possible to examine the execution dependencies within an application at weave time, and then use this dependency graph to discern sets of join points that are potentially within the control flow of an explicitly matching join point, i.e., it is possible to establish the set of *potential implicitly matching join points.* The weaver can then modify the code of the potential implicit matches to include a runtime check to see if they are in the `cflow` and to apply advice accordingly.

Two algorithms for finding potential implicitly matching join points at weave time were examined during weaver design: one naïve and the other fine-grained. The *naïve algorithm* treats every join point in an application as a potential match. From the code generation point of view, the consistency of this algorithm makes it easy to implement. This algorithm is naïve, because the set of potentially matching join points is not optimized. If the `cflow` primitive pointcut designator is not further constrained within the pointcut, it is very easy for the aspect programmer to introduce significant performance overhead. For instance, all field accesses within an application not specifically ruled out by the designators combined with `cflow` would include a dynamic check that adversely affects execution time.

A *fine-grained algorithm* helps in large scale applications or where performance is an issue by referencing a dependency graph when deciding on potentially matching join points. With this algorithm, only join points whose implementation is activated directly or indirectly by an explicitly matching join point need to be tested at runtime to see if it is in the `cflow`. This dependency graphed can be based on calls or invocations. The call graph maps the activation of one join point by another. Each join point implementation is a node and any direct activation of another join point implementation appears as a directed arc to the node corresponding to the implementation of the join point being activated. The invocation graph reflects which join point implementations activate the implement of a node, because the directed arcs point back to the calling join point's implementation. With either graph, a join point should be tested to see if it is in the `cflow` if there is a path in the graph from the implementation of an explicit match to the implementation of the join point being examined.

Of the two algorithms, the naïve algorithm is most suitable for our prototype in its current state as it is significantly simpler to implement. Our decision is consistent with the commercial implementation of AspectJ, which also uses a naïve algorithm [Hil'04].

4.3.3.2.3 Designing Runtime `cflow` Checks

At runtime, the *cflow active flag* corresponding to a `cflow` primitive pointcut designator determines if a join point explicitly matching that cflow is active. When the flag is active, implicitly matching join points should be influenced by advice associated with the pointcut containing the `cflow` designator. In the example below, written in AspectJ syntax, a call to `Stack.Push` would cause the flag associated with the `cflow(execution(* Stack.Push(*)))` to become active, which signals that advice associated with the `cflow_or_exec` pointcut is executed for implicitly matching join points.

```
cflow_or_exec() : cflow( execution(* Stack.Push(*))) ||
                  execution(* Stack.Pop(*));
```

`cflow` activation has to be determined on a thread by thread basis, as the activation of an implicitly matching join point is dependent on how that join point was reached during execution, and such paths will exist on a thread by thread basis. With the weaving library, the *cflow active flag* is implemented by a `ThreadStatic` field in a type corresponding to the `cflow` pointcut name and aspect name. To allow it to be `ThreadStatic`, the field must also be a static field. The field itself is more of a reference counter than a flag. It is implemented as an integer that is incremented each time an explicitly matching join point corresponding to the `cflow`'s argument is started and decremented each time an explicitly matching join point is ended. This makes it simpler to set and reset the flag in light of recursion.

4.3.3.2.4 Difficulties with Logical Not Due to `cflow`

Join point matching involves examining a join point to see if it matches a pointcut specification that consists of logically combined primitive pointcuts designators with the logical not operator applied at any point in the composition. An example of a pointcut written in AspectJ syntax is shown below and it selects all join points except intra type calls made by methods in type `FooType`.

```
Simple_match() : !(call(* FooType.Foo(..) || within(FooType))
```

A naïve implementation of join point matching is to perform a depth first, left to right traversal of a graph built from the pointcut specification at weave time. Leaf nodes correspond to primitive pointcut designators, and internal nodes to logical operators. Leaf nodes that do not match the join point should return false, and the logical operators are applied to the Boolean results of children nodes. Take for example the call to `Foo` from the `FooBar` method in following source:

```
class BarType {
      void FooBar() {
            Foo();
      }
      ...
}
```

Application of the algorithm to the `Simple_match` named pointcut would result in the graph shown in Figure 4.13, in which the results for applying the graph to the code above are noted at each node in the figure.

On the one hand, this algorithm has the advantage that its control structure matches the layout of data, which is advocated as good programming practice [Fri'01]. However, the algorithm is naïve in that it does not deal with potential matches identified at weave time by `cflow` pointcuts. Take the following pointcut as an example.

```
Cflow_match() : !(cflow(call(* FooType.Foo(..)) && within(FooType))
```

Blindly applying the negation, as shown in Figure 4.14, will result in the wrong answer. When the naïve algorithm attempts to match the `Cflow_match` named pointcut to the previous code, the result is wrong. Instead of the logical not inverting the result, it should invert the runtime value of the `cflow` active flag for which the join point match would occur.

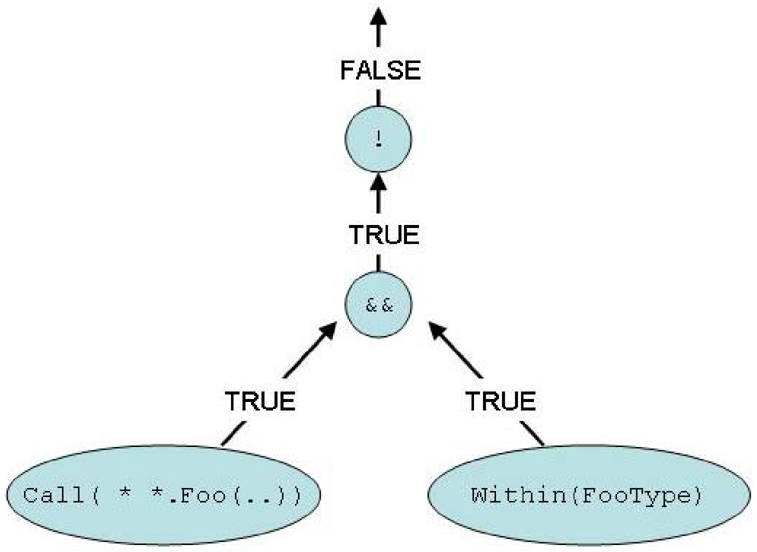


Figure 4.13: Successful application of naïve join point matching.

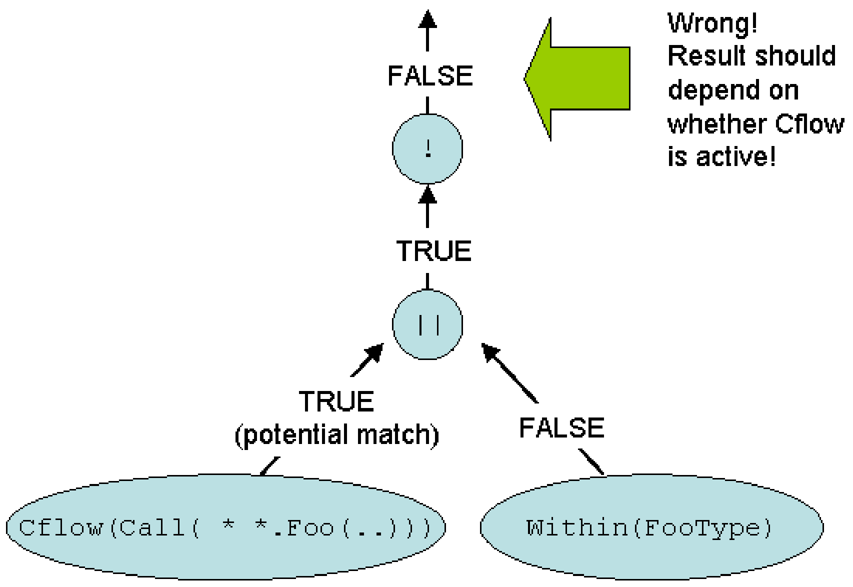

FALSE
!
TRUE
||
Wrong! Result should depend on whether Cflow is active!
TRUE (potential match)
FALSE
Cflow(Call(* *.Foo(..)))
Within(FooType)

Figure 4.14: Unsuccessful application o f naïve join point matching.

To account for `cflow` matching, we distribute the logical not operator across each individual primitive pointcut designator in a pointcut. This distribution is done according to the rules of Boolean logic, and the distribution allows each primitive pointcut designator to decide on the semantics of the logical not. This allows application of logical not to the `cflow` and `cflowbelow` join points to revise the value of `cflow` active flag for which advice execution is triggered.

4.3.3.3 Implementing the `IAspect` Interface

Recall from Figure 4.6 that the `IAspect` interface implementation provided by the weaver is implemented in the `TCD.DS.DSG.Weave.Reflect.Aspect` class. An outline of this class is shown in Figure 4.15. The ability to use the AspectJ's reflective types for describing join points is based on the availability of and Java interpreter for the CLI in the form of J# [Mic'04e]. Thus, the AspectJ classes that support the keywords `thisJoinPoint`, `thisJoinPointStaticPart` and `thisEnclosingJoinPointStaticPart` have been partially ported to the CLI. The port is partial in that not all of the features of the reflective objects are supported. Also, the weaver does not predetermine whether or not reflective objects are required, so the reflective objects must always be initialised if advice inherits `TCD.DS.DSG.Weave.Reflect.Aspect`.

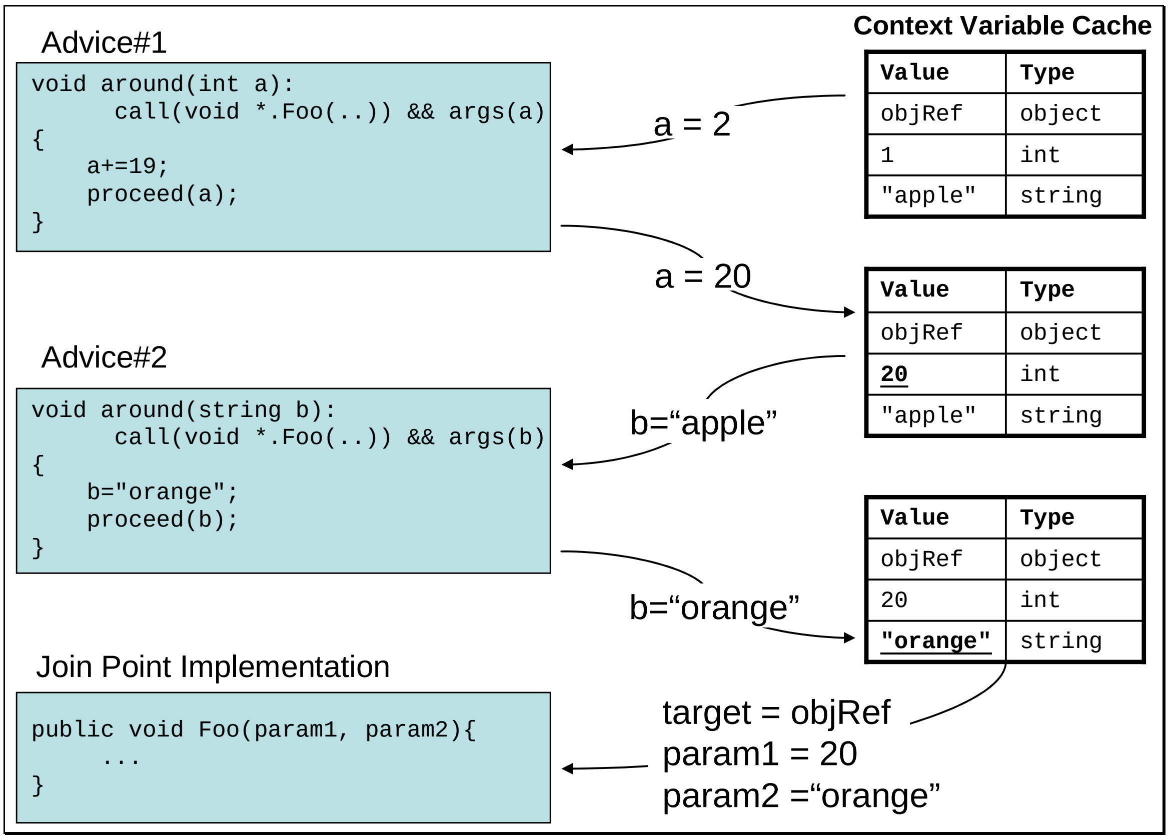
```
namespace TCD.CS.DSG.Weave.Reflect {

public class Aspect : IAspect {

  org.aspectj.lang.JoinPoint              IAspect.JoinPoint{ ... }
  org.aspectj.lang.JoinPoint.StaticPart   IAspect.JoinPointStaticPart{ ... }
  org.aspectj.lang.JoinPoint.StaticPart IAspect.EnclosingJoinPointStaticPart{ }

  object IAspect.Proceed(params object[] obj) { ... }


  // Prototype-specific extensions
  object[] IAspect.ProceedParams { ... }
  bool  IAspect.ExecJoinPointRecurseActive { ... }

  System.RuntimeMethodHandle IAspect.JoinPointInvocationCache{ ... }

  void IAspect.QueueAroundAdvice(int[] mappings, RuntimeMethodHandle
                                 adviceMeth, RuntimeFieldHandle[]
                                 cflowActiveFlagRefs,
                                 bool[] cflowActiveFlagStates){
      ...
    }
  }
}
```

Figure 4.15: Prototype implementation of `IAspect` interface in Figure 4.6.

The implementation of `Proceed` uses a dynamic queuing mechanism that allows the sequence of calls corresponding to the `Proceed` invocations to be set up on a join point by join point basis. The dynamic queue is introduced, because separate compilation of aspect and component means that the methods invoked by `Proceed` cannot be known at compile time. The implementation of this dynamic queuing mechanism is the subject of the following subsection.

4.3.3.3.1 Designing the `Proceed` Method for Around Advice

Central to the ability of a `Proceed` implementation to call advice methods or the join point direct invocation method is the presence of a cache of all the context variables for the join point at which advice is executing. This cache is the source for parameters of the method executed in response to `Proceed`, be it another advice method or the direct invocation method for the join point. Since advice may change join point context variables, the cache must be kept up to date with the parameters used to call `Proceed`. A demonstration of cache updating is shown in Figure 4.16. In this figure, two pieces of around advice, written in AspectJ syntax, are applied to join points corresponding to the execution of a method `Foo`. Since `Foo` is an instance method, the cache contains a reference of type `object` used to invoke the method. Also, the cache contains the parameters for method execution, which are of type `int` and `string`. The first cache update is caused by the `Proceed` call made by Advice#1, which updates the value of the integer argument of the join point from 1 to 20. The cache is updated again in Advice#2 after `Proceed` call is made. This time the second argument of the join point is updated to "`orange`".



```
void around(int a):
      call(void *.Foo(..)) && args(a)
{
    a+=19;
    proceed(a);
}
```

Value	Type
objRef	object
1	int
"apple"	string

```
void around(string b):
      call(void *.Foo(..)) && args(b)
{
    b="orange";
    proceed(b);
}
```

Value	Type
objRef	object
20	int
"apple"	string

Value	Type
objRef	object
20	int
"orange"	string

```
public void Foo(param1, param2){
     ...
}
```

Figure 4.16: Join point execution context cache updates during around advice execution.

Our implementation of `Proceed` uses one parameter cache per thread, which is consistent with the decision not to match advice execution with pointcuts. Were around advice applied to around advice execution join points, two sets of context parameters would have to be cached: one set for the advice execution join point and another for the component join point at which advice was executing. This situation is presented graphically in Figure 4.17. Eventually, the number of caches would have to be variable to allow multiple applications of around advice to around advice execution join points. By not matching advice to pointcut, we avoid this scenario.

```
Method FooClass.Foo
 (execution join point)
        |
        | --> around advice of FooAspect
        |           |
        |     (cache Foo parameters in
        |      FooAspect.ProceedParams)
        |           |
        |           |
        |           | --> Method FooAspect.Bar
        |           |      (implements around advice)
        |           |           |
        |           |           | --> around advice of FooAspect
        |           |           |           |
        |           |           |     (cache Bar parameters in
        |           |           |      FooAspect.ProceedParams)
        |           |           |           |
        |           |           |           |
        |           |           |           X
        |           |           |
        |           |           |      Previously cached parameters
        |           |           |      are wiped out!
```

Figure 4.17: Failure with using single cache for join point context variables when around advice is applied to advice execution.

The dynamic queuing mechanism is responsible for caching references to methods that will be called by `Proceed` as well as caching mappings between `Proceed` parameters and join point execution context. The method queue consists of reflective objects from the `System.Reflection` API describing the around advice methods to be called as well as the join point's direct invocation method. Any particular around advice method may chose not to make a `Proceed` invocation; however, the full sequence of around advice methods matching a join point along with the direct invocation method for that join point are queued anyway. The queue is generated at runtime by code added by the weaver. Code to add methods to the queue is emitted each time an around advice matching the pointcut is found by the aspect modelling system. After all pointcuts are considered, a reflective object describing the direct invocation method is added to the queue. Advice methods dependent on `cflow` join points are queued alongside details of the `cflow` invocation flag and the flag state that triggers advice execution. This allows `Proceed` to make a dynamic check to determine whether or not the advice should be applied based on whether the `cflow` is active or not.

Proper updating of the join point context as well as proper invocation of advice is a matter of knowing which `Proceed` parameter corresponds to which join point context variable. These mappings are cached at the same time as the sequence of advice methods and the direct invocation method are queued. A mapping is succinctly described by an integer array in which the element index corresponds to a parameter passed to `Proceed` and the element data indexes a join point context variable in the join point execution context cache. So, for around advice that is applied to join points specified by the following pointcut,

```
  pointcut PushExec(StackElement obj): execution( * Stack.Push(*))
                                       && args(obj);
```

the `Proceed` call would look like for the following:

```
      Object[] tmp = new Object[1];
      tmp[0] = (Object)stackElement;
      Proceed(tmp);
```

and the corresponding description of `Proceed` parameter to join point context mappings would be as follows:

```
      int[] proceedMap = { 1 };
```

So that the first `Proceed` parameter, held in element 0 of its `object[]`, is mapped to the second parameter in the join point execution context cache.

4.3.4 Limits to the Weaving Library

The following section identifies limits to the weaving library described in the previous subsections. The weaver is limited due to using a one-pass code generation system and due to statically checking for matches between join points and pointcuts. Another limiting factor has been the developer resources available for its implementation and testing. Thus, the library does not support multi-aspect weaving, limits the advice types and join point types supported, and does not allow pointcut matching for join points implemented with instructions that are parameterised by references placed on the stack at runtime.

4.3.4.1 Weaving API

The weaving API only provides a method to weave a component against one crosscutting specification. The objectives for this prototype are to support attribute-based property selection and language-independent weaving. Supporting multiple XML files is a matter of revising the interface to accept an array of XML file references. Although modifications of the weaver implementation would be required, the architecture will scale to support multiple XML files with multiple aspects. The same is true for supporting multi-module assemblies.

Likewise, a particular XML-based crosscutting specification can only contain one aspect. The system makes no effort to resolve the “composition problem” [Szy'02] that refers to problems that arises when dependencies between aspect types restrict their ability to be composed with the same component.

4.3.4.2 Aspect Model Limits

Our prototype limits the kinds of after advice to after returning advice. Our design does not exclude after and after throwing advice, but for the purposes of experimentation these kinds of advice were not required. This is a pragmatic solution to reducing the amount of programming required to implement the weaver. Code generation for after advice and after throwing advice requires try/finally blocks to intercept exceptions, while no such restrictions apply to after returning advice. Another limitation to reduce complexity is to limit advice method look up to take into account only the method name. Thus, the prototype weaving library looks up advice methods by name only, i.e., parametric overloading of advice implementations is not implemented.

As with the AspectJ compiler, our prototype has placed restrictions on handler join points so they cannot be influenced by around advice. This is restriction is in place, because advice applied to exception handlers does not suit the design of direct invocation methods. The direct invocation method will want to jump to the body of the execution handler when `Proceed` is called; however, it is not in keeping with the concept of exception handling blocks to jump directly into an exception handling block.

The prototype supports the full range of typed formal parameters that can be exposed using `args`, `this` and `target` primitive pointcut designators, but the prototype weaver lacks error checking to weed out invalid context exposure. For example, when a static method is being called there is no check to make sure the `target` context variable is not accessed.

Object pre-initialization join point types are not supported. Recall that object pre-initialization join points correspond to call join points executed during object construction whose results are used in calls to other constructors or to a superclass constructor. Object pre-initialization join point types cannot be detected in a single pass weaver. They appear as call join points until a constructor call appears that uses their result as a parameter, and by that time the pre-initialization join point has been emitted to the woven assembly being created by the weaver. Fortunately, object pre-initialization join points are rarely used [Lad'03].

As for thread safety, there has been an effort to make the system thread safe in so far as the reference counting required to properly implement `cflow` and `cflowbelow` pointcuts is concerned. However, thread safety has been a requirement for other features that require the dynamic context be tracked such as initialization join point support, and in these instances thread safety was not tested.

4.3.4.3 Unsupported opcodes

The join points that can be properly woven by the weaving library are limited according to the opcode used to implement them. With respect to call join points, there is an issue as to what instruction is used to make the call. `Calli` is not properly supported. `Call` and `Callvirt` take metadata tokens as their parameters, so the target method for the call is clearly identified from the call instruction. `Calli` is different in that it draws from a function pointer on the stack, and so a dynamic check against all pointcuts is required to determine whether or not the method invocation should be influenced by advice. However, the current weaver only does static comparisons against pointcuts. Finally, only the CLI's standard calling convention is supported in which the arguments of a method are pushed on the stack in their order of appearance in a C# method declaration starting with the object instance reference if required. In contrast, methods in CLI assemblies can be defined with platform specific calling conventions. These other conventions would require updates to the call parameter caching code so that the weaver knew where on the stack each parameter is located.

With respect to field access join points, some implementations require a runtime check to determine which pointcuts they match. Field accesses that are made with `Ldfld`, `Stfld`, `Ldsfld` and `Stsfld` instructions name their targets with metadata tokens at compile time, and these metadata tokens identify the field's signature. Thus, pointcut matching can be done at compile-time. However, there are CIL instructions that obtain their targets from the stack. For example, `Stelem_Ref`, and `Ldelem_Ref` allow access to array elements based on an index and a reference to the containing array. If we take the AspectJ implementation as precedent, it would seem that these instructions are not supported as call join points. Take the example of trying to identify an array element access using a pointcut. For AspectJ V1.0.6, there is no syntax for accessing a particular element of an array field. In addition to difficulties determining the array field being accessed, the CLI allows fields to be accessed from an address stored on the stack. Specifically, `ldflda` and `ldsflda` instructions place field addresses on the stack, and the addresses can be used to access data from this instance using a `ldobj` instruction. Although the data type being accessed is known, other details such as the containing type are harder to determine in the one pass weaver used by Weave.NET. Moreover, the AspectJ V1.0.6 semantics that the weaver library implements are targeted at Java, and they have nothing to say about field address access. Thus, we have chosen to not support join points corresponding to the execution of these instructions along with others that deal with data in terms of addresses.

4.4 Integrating with the Execution Environment

The prototype weaver does not modify the existing class loader infrastructure of a CLI installation, instead weaving is implemented by a program that intercedes at execution start up time. Rather than launching the CLI normally, a batch file named `weave.net.bat` is called with the name of the assembly containing the application entry point. This batch file starts up the execution environment, and calls a program to discover aspects and components and weave them. The weaving program expects all aspects to appear in a subdirectory named `aspect`, and all components to appear in a subdirectory called `component`. This organisation of files should be done by the application deployer role. Each component is woven against crosscutting specifications in the XML files appearing in the aspect directory. After weaving is complete, the CLI's reflective API is used to launch a new CLI execution environment, called an AppDomain, using the entry point of the component passed when the batch file was called.

4.5 Summary

In this chapter we presented details of a prototype weaver that supports the aspect-based property programming model for the CLI platform. The CLI platform provided considerable infrastructure in the form of specifications for language-independent component development, attribute type implementation and language-independent annotation of component source with attribute types. In addition, APIs for code generation, reflection and XML processing facilitated weaver implementation. The XML schema for aspect-based property pointcut-advice semantics was derived systematically from a BNF specification of the pointcut advice semantics of AspectJ V1.0.6 according to conversion rules that were presented in section 4.2. The weaver, Weave.NET, relies on a weaving library that implements two APIs. The `TCD.CS.DSG.WeaveDotNet` class provides a static method that composes a component and an aspect according to the aspect's XML-based crosscutting specification. The `TCD.CS.DSG.Weave.Reflect.Aspect` class provides an implementation of a modified `IAspect` interface with support for `Proceed` calls and join point reflection. The `IAspect` methods are separate from the type defined by `Aspect` so that `Aspect` may be inherited by types implementing aspect behaviour without concern for naming conflicts. The weaver consists of two systems: a code generation system that organises byte code instrumentation of components, and an aspect modeling system that determines the bindings between join points and advice. The details of join points are encapsulated by the code generation system and passed to the aspect modeling system, which determines the join point to advice bindings. Byte code is added to the component being woven to transfer control to advice as required, to support dynamic decision making required to detect join points in a `cflow` and `cflowbelow` pointcuts, and to initialise data structures used by the `IAspect` implementation. The weaver is limited due to using a one-pass code generation system and due to statically checking for matches between join points and pointcuts. The weaver intercedes at start up by having the application deployer launch execution environment with a special batch file. This batch file launches a weaving program that determines the aspects and components to weave based on directory location. After weaving components, it passes control back to the entry point of the application being executed.

Chapter 5 Evaluation

“L'implementation c'est bien mais l'évaluation c'est mieux”
–Jean-Marc Seigneur

In this evaluation, aspect-based properties supported by Weave.NET, our prototype weaver, are tested for language-independence and compared with functionally equivalent context properties.

The language-independence challenge involves writing component types in C# [Ecm'03a], VB.NET [Mic'04d] and SML.NET [Ken'03] and binding each of these to three logging aspects, also written in each of these languages. This selection of languages provides representatives from the object-oriented, procedural, and functional programming paradigms. In the first instance of this test, we use custom crosscutting to establish interoperability of components and aspects written in different languages. In the second instance of this test, the custom crosscutting is rewritten in terms of attributes to establish that attribute-based property selection does not conflict with language-independence.

Comparisons with existing contextual composition involve writing context properties and aspect-based properties to address a selection of crosscutting concerns. Comparisons are made with CLR contexts. CLR contexts are an implementation of extensible contextual composition for the CLI. Qualitative differences in the implementation and application of crosscutting functionality are examined in the context of the task of creating a profiling property to measure method execution time. Using the profiling property, we can examine differences in execution overhead when logging is applied to a recursive Fibonacci series element generator using a context property and then again as an aspect-based property.

Finally, context properties and aspect-based properties are applied to the task of implementing a memoization optimization for an existing component for which source code is unavailable. Memoization [Men'97] is a term that describes result caching for the purposes of enhancing performance. In our case, this optimization is applied to improve the execution performance of the recursive Fibonacci series generator.

We find that using aspect-based properties to implement crosscutting concerns avoids preplanning and tailorability issues of context properties, that aspect-based properties are lighter weight than context properties, and that aspect-based properties are simple to adopt. Migration considerations are addressed by the language-independence characteristics of the programming model used with aspect-based properties and the ability to reuse aspects without modifying their crosscutting specifications. Aspect-based properties can be used by component and application developers without the need to adopt new programming languages. Also, aspect-based properties can be bound to existing components regardless of implementing language of either aspect behaviour or the component. In terms of reusability, the attribute-based property selection used with aspect-based properties is consistent with language-independence in that attributes do not dictate programming language, and attributed-based property selection avoids the need to revise crosscutting specifications. In the comparison tests we noted that unlike context properties, aspect-based properties did not place architectural constraints on the components adopting aspect-based properties. Application of logging and timing written as context properties required substantial changes to component architecture, and for memoization these changes were unrealistic. In contrast, aspect-based properties could be adopted in each test case without the need to revise component implementation. Furthermore, the execution overhead introduced by aspect-based properties is one tenth that of context properties, and the implementation of aspect-based properties was more succinct and articulate than the context-based equivalent.

Indirectly, this evaluation uncovers issues with aspect-based properties with respect to the specification of crosscutting and the implementation of aspect-component composition with the Weave.NET weaver. The use of XML simplifies the parsing and validation of crosscutting specifications, but observations from the evaluation process indicate that writing XML by hand is error prone. As described in Chapter 3, XML uses CLR type names rather than language-specific monikers. In our tests we noted that mapping type specifications between different programming languages and the CLR is not always intuitive. Also, our prototype weaver has difficulties when multiple aspects are woven into an application. While the weaver can only weave one aspect per component, we were expecting that multiple aspects could be supported provided they were woven into different components. Due to our implementation, weaving for one property can inadvertently load assemblies into the execution environment, and once a component is loaded, aspect-based properties can no longer be woven to that component. Finally, the quality of our weaver draws attention to the fact that developing a commercial grade weaver requires significant programming resources that are beyond the scope of this project.

The evaluation is broken into three sections. Section 5.1 examines migration issues with subsections focusing on the language-independent nature of aspect-component composition and attribute-based property selection. Section 5.2 contains comparisons between crosscutting functionality implemented with context properties and aspect-based properties. Finally, section 5.3 summarises usability issues encountered during the evaluation of the programming model and the Weave.NET implementation.

5.1 Migration Path Feasibility

In our view, the primary difficulty with adopting aspect-based technology is the preservation of existing components, development tools and developer knowledge. Traditionally, AOP has forced the user to adopt a particular language for the implementation of component types that will be woven with aspects [Laf'03]. Unfortunately, this is not feasible where component source is not available or where a large code base exists in a language without commercial AOP support. Such situations are in sufficient abundance that there has been a recent recognition of the need for a focus on language-independence in the AOP community [Sab'04] with the goal of allowing component developers to use AOP with existing components, an existing component code base and existing component development languages. Thus, the primary issue in adoption is language-independence. Language-independence addresses the need to make aspect-based technology compatible with existing development technology. We can also look at preservation of developer knowledge from the point of view of wanting to avoid new technologies that require upgrades to developer skills. With aspect-based properties, we want to hide the aspect-oriented mechanisms from the parties involved in development as much as possible. While it is not possible to hide AOP mechanisms from the aspect developer, it is possible to hide them from the component developer using attribute-based property selection. Thus, the secondary issue in adoption is that aspect reuse be available in terms of attribute-based property selection, which avoids the need to modify aspects and which is consistent with language-independence.

5.1.1 Language-Independent Composition

Evaluation of language-independence is a matter of implementing aspect-based properties in a variety of languages and applying them to components also written in a variety of languages. The test of language-independence is discussed in section 5.1.1.1. In this evaluation, components implement an algorithm that enumerates Fibonacci series elements. The demonstration of AOP techniques using a Fibonacci series enumeration algorithm is quite common as pointed out in [Cos'03]. Indeed, the algorithm has become common place in demonstrating AOP concepts in commercial tools such as AspectWerkz [Bon'04a]. Custom crosscutting is used to bind components and aspects in this test. Recall from Chapter 3 that custom crosscutting is used by the application integrator role of the aspect-based property programming model to combine aspects with components not already annotated with attribute types. The evaluation is concerned with demonstrating the binding of aspect-based properties to components across language barriers and not with demonstrating the clear-box crosscutting available with aspect-based properties. For this reason, the aspect-based property is limited to logging execution join points as opposed to call or field access join points. Components implementing the Fibonacci series algorithm as well as those implementing the behaviour of the logging aspect-based property have been written in three CLI-producer compliant languages that were selected in order to have examples that include representatives from multiple programming paradigms. C# provides an example of a mainstream object-oriented language, as it shares many similarities with Java and C++. VisualBasic.NET (VB.NET) is chosen as a representative of procedural programming languages. Despite having adopted significant OO extensions in this latest update, VB.NET still provides procedural programming capabilities, and its existing base of users are focused on procedural-style programming. Finally, SML.NET provides functional programming for the CLI. In order to allow attribute-based property selection in the second set of tests, we have limited ourselves to specifying our SML.NET algorithm in terms of a class type for which the SML.NET compiler provides attribute support.

In the course of implementing the evaluation, we noted that writing custom crosscutting is error prone due to difficulties with writing the crosscutting specifications and in advertent join point selection. The causes of these problems are discussed separately in section 5.1.2.2.

5.1.1.1 Verifying Language-Independence with Custom Crosscutting

The target for our language-independence tests is a component implementing a recursive algorithm that enumerates members of the Fibonacci series. The algorithm, shown in Figure 5.1, includes two methods, one that generates elements in the Fibonacci series, and a second that reports a series of elements generated using the former method. Components containing these methods have been written in C#, VB.NET and SML.NET. The C# version shown in Figure 5.1 is typical of the algorithm, which is recursive regardless of the programming language used.

```
public class FibonacciSeries
{
  public void FibSeries(int seriesLen)
  {
    for (int i = 0; i<= seriesLen; i++)
    {
      long result = Fibonacci(i);
      System.Console.WriteLine("Element \t"+ i+ "\tvalue \t"+result);
    }
  }

  public long Fibonacci(int n)
  {
    if (n > 1)
      return this.Fibonacci(n-1) + this.Fibonacci(n-2);

    return 1;
  }
}
```

Figure 5.1: C# source algorithm to enumerate Fibonacci series elements.

Our Fibonacci algorithm lacks an explicit indication of its complexity, but this is remedied by adding a logging aspect-based property that reports the start and end of execution join points. Normally only one implementation of logging would be need, but given our focus on language-independence, implementations of the logging aspect-based property are created for each of the three test languages. Writing the aspect-based property in a particular language involves implementing the aspect behaviour in that language. Logging is bound to a component using custom crosscutting as if binding were written by an application integrator. Logging is a fairly simple concept made simpler by limiting the aspect-based property to reporting the start and end of a method execution to the application console rather than logging to a file. A sample implementation of logging behaviour is shown in Figure 5.2 written in SML.NET this time. The method names in the source allude to the kind of advice they implement. The appearance of multiple methods with the prefix `LogAfterJoinPointXXX` methods alludes to difficulties with supporting after advice for different return types, which is discussed shortly.

```
structure Aspect_ML_Logging =
struct
_classtype Logger() : TCD.CS.DSG.Weave.Reflect.Aspect()
 with
  LogBeforeJoinPointInt (param:int) =
  let
   val jptInfo = valOf(this.##get_JoinPointStaticPart());
  in
   print "Join point: "; print (valOf(jptInfo.#toShortString())); print "\n";
   print "Execution parameter: "; print (Int.toString(param)); print "\n"
  end
 and
  LogAfterJoinPointLong(param:int, result:Int64.int)=
  let
   val jptInfo = valOf(this.##get_JoinPointStaticPart());
  in
   print "Join point: "; print (valOf(jptInfo.#toShortString())); print "\n";
   print "Execution parameter: "; print (Int.toString(param)); print "\n";
   print "Execution result:    "; print (Int64.toString(result)); print "\n"
  end
 and
  LogAfterJoinPointVoid (param:int) =
  let
   val jptInfo = valOf(this.##get_JoinPointStaticPart());
  in
   print "Join point: "; print (valOf(jptInfo.#toShortString())); print "\n";
   print "Execution parameter: " ; print (Int.toString(param)); print "\n";
   print "Execution result:    NONE!"; print "\n"
  end
 end
end
```

Figure 5.2: Implementation of logging behaviour written in SML.NET

The aspect-based property's custom crosscutting specification is generally the same regardless of the language implementing aspect behaviour and the component to which the aspect is applied. The signatures of methods that implement aspect advice have been purposely made the same in each implementation of the logging aspect. This limits the need to modify the pointcut specification, which is shown in Figure 5.3, depending on the language implementing aspect-based property behaviour. The crosscutting specification shown in Figure 5.3 defines a named pointcut called `SomeMethodExecution` that identifies method invocations that take an integer as a parameter regardless of the return type. The slight variation in the XML specifications used by each aspect-based property comes from the type name identifying aspect behaviour. In contrast to VB.NET and C#, the SML.NET aspect behaviour is exported as a nested type, whose type signature includes the name of the enclosing class. The custom crosscutting specifications are reusable without modification in that they can be applied to components without change. Reuse then relies on the component's types being the same in terms of members and member signatures regardless of implementing language.

```
<item>
  <named_pointcut>
    <modifier><public/></modifier>
    <name>SomeMethodExecution</name>
    <local_var_ref>
      <var_type>Int32</var_type>
      <var_name>data</var_name>
    </local_var_ref>
    <pointcut>
      <and>
        <pointcut><primitive>
          <execution>
            <method_signature>
              <return_type><type_name>*</type_name></return_type>
              <join_point_type><type_name>*</type_name></join_point_type>
              <method_name>*</method_name>
              <parameters>
                <parameter><type_name>Int32</type_name></parameter>
              </parameters>
            </method_signature>
          </execution>
        </primitive></pointcut>
        <pointcut><primitive>
          <args>
            <parameter>
              <formal_parameter_name>data</formal_parameter_name>
            </parameter>
          </args>
        </primitive></pointcut>
      </and>
    </pointcut>
  </named_pointcut>
</item>
```

Figure 5.3: A pointcut identifying method execution join points to which logging is applied.

Logging is applied to execution join points using separate before and after advice corresponding to the start and end of each execution join point, but it is the after advice that highlights compromises made in the design of Weave.NET. The use of the before advice in Figure 5.4 and the after returning advice in Figure 5.5 is chosen to provide contrast with other evaluation tests later in this chapter, which focus on around advice. The application of before advice is straightforward. Our join points all take the same argument, so one before advice can be used if logging is to access join point execution state through typed formal parameters. The variation in return types of execution join points in our `FibonnacciSeries` join points is accounted for with different kinds of after advice, one for each possible return type. For code clarity, it would be preferable if methods implementing logging-related after returning advice where identically named. Unfortunately, CLR allows method name overloading on the basis of parameter type, but not on the basis of return type. As an alternative, our solution uses a crude form of mangling in which methods implementing logging-related after returning advice carry a suffix corresponding to their return type. Another solution would be to provide a single after returning advice that returned an `Object` reference. While the `Int32` and `Int64` return types would match, we would still need separate advice to influence methods that have a `Void` return type. Note that in the version of the prototype weaver used for evaluation, fully qualified type names were not required, thus the crosscutting specification uses the type name `Int32` and not `System.Int32`. We discuss the purpose of this change later in this section.

```
     <!-- before advice, where single argument is a 32 bit int -->
     <item>
       <advice>
         <before>
           <formal_param>
             <var_type>Int32</var_type>
             <var_name>data</var_name>
           </formal_param>
           <pointcut><primitive><pointcutId><name>
             SomeMethodExecution
           </name></pointcutId></primitive></pointcut>
           <behaviour><name>LogBeforeJoinPointInt</name></behaviour>
         </before>
       </advice>
     </item>
```

Figure 5.4: Before that applies logging before join point execution.

```
     <!-- after returning advice, where result is an 64 bit int -->
     <item>
       <advice>
         <after>
           <returning_params>
             <var_type>Int64</var_type>
             <var_name>data</var_name>
           </returning_params>
           <pointcut><primitive><pointcutId><name>
             SomeMethodExecution
           </name></pointcutId></primitive></pointcut>
           <behaviour><name>LogAfterJoinPointLong</name></behaviour>
         </after>
       </advice>
     </item>
     <!-- after returning advice, where result type is void -->
     <item>
       <advice>
         <after>
           <returning_params>
             <var_type>Void</var_type>
             <var_name>none</var_name>
           </returning_params>
           <pointcut><primitive><pointcutId><name>
             SomeMethodExecution
           </name></pointcutId></primitive></pointcut>
           <behaviour><name>LogAfterJoinPointVoid</name></behaviour>
         </after>
       </advice>
     </item>
```

Figure 5.5: After advice that applies logging after join point execution.

The actual language-independence test involves weaving a new assembly for each combination of component and aspect and verifying it to be valid. Nine combinations of component and aspect exist, and these combinations are identified in Table 5.1. For each combination of component and aspect woven, the result will be a new version of the component assembly that has been modified to access logging functionality from the aspect being woven. At issue is whether this new assembly is a valid CLI component. To verify validity, the new assembly is examined in two ways. First, the new component is checked programmatically to verify that the CIL and metadata meet type safety requirements. Second, the new component is executed. The expectation is that the assembly will execute and exit without generating an exception or error. As well, we look for logging functionality to be consistent with other components using the same aspect implementation. Component verification is performed using Microsoft's Peverify [Mic'04c], which is an automated tool for type safety verification that reports problems with a component's metadata as well as the CIL of its methods. Proper execution involves piping console output to a file and then examining the results to ensure each of the nine executions generate the same results.

Note that the bootstrap loading mechanism described in section 4.4 of Chapter 4 is not used during this or any other testing. The bootstrap loader performs all weaving before application start up. The difficulty with this approach is that it prevents us from isolating weaving from the rest of the application. So rather than use the bootstrap loader, all our tests call the weaving API directly.

Table 5.1: Combinations of component and aspect woven according to implementing language.

		Aspect Type Language		
		C#	**VB.NET**	**SML.NET**
Component Type Language	**C#**	C# + C#	C# + VB.NET	C# + SML.NET
	VB.NET	VB.NET + C#	VB.NET + VB.NET	VB.NET + SML.NET
	SML.NET	SML.NET + C#	SML.NET + VB.NET	SML.NET + SML.NET

The resulting weaving times for each test combination in Table 5.1 are shown in Table 5.2. The figures are the average amount of time that the weaver spent processing a component, and the average is from figures collected from three trials. Table 5.2 also indicates the number of instructions processed in the component being woven. As noted in Table 5.2, execution times are representative of a debug build running on a 497MHz Pentium III laptop with 384Meg of RAM under WindowsXP Professional. Results are established by inspecting the laptop's high speed timer which operates at approximately 3.6 MHz. The times are representative in that the trials are not rigorous and simply aim to establish the ability to weave across language boundaries.

Although not shown in Table 5.2, the static overhead of loading types required by the weaver is significant. Our initial tests discovered a substantial overhead in the first weaving operation of any test group, and so we added a dummy weave that applied a logging aspect against a minimally sized component, i.e. one containing a single method of 3 CIL instructions in size. From this we were able to determine an overhead of approximately 2000 milliseconds was involved in loading the components involved in weaving for the configuration used in Table 5.2.

Table 5.2: Weave times for combinations of components and aspects written in a variety of languages.

		Aspect Type Language		
		C#	VB.NET	SML.NET
Component Type Language	C# (112 instructions)	160 ms	169 ms	169 ms
	VB.NET (131 instructions)	110 ms	106 ms	105 ms
	SML.NET (444 instructions)	168 ms	165 ms	151 ms

ms – milliseconds or 10^{-3} seconds

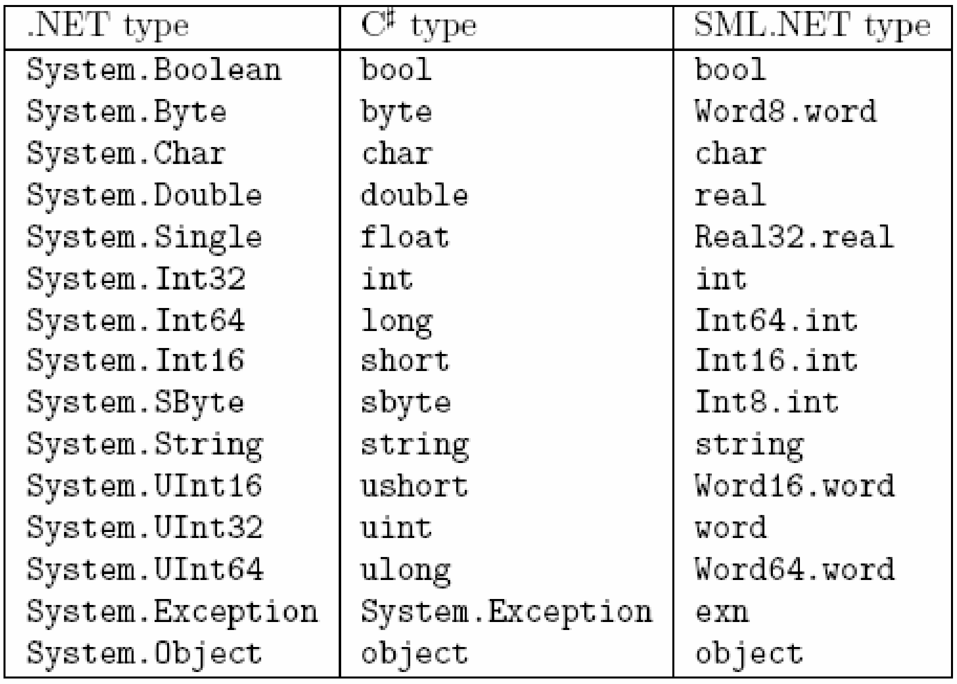
Trials performed with debug build on 497MHz PentiumIII laptop with 384Meg of RAM under WindowsXP

The language-independent custom crosscutting tests allow us to verify that aspect-based properties can be implemented using functional, procedural, and object-oriented programming languages and woven with components implemented in any of these languages.

5.1.1.2 Problems with Language-Independent Custom Crosscutting

In implementing the weaving trials, we noted that the specification of types in XML is not as straightforward as it could be, because mapping from language-based type names to CLI type names must be done manually. Writing a custom crosscutting specification in XML involves using metadata descriptions to select join points. As pointed out in Chapter 3, the crosscutting semantics of aspect-based properties are specified in terms of CLI types, and not the development language types with which a programmer will be familiar. The need to map from development language types to CLI types is acute in the case of primitive types, whose CLI names vary considerably from those used in the source code of a component. For example, Table 5.3 shows the mappings between SML.NET primitive

Table 5.3: Mapping between CLI (.NET) types and C# / SML.NET equivalents, taken from [Ken'03].



.NET type	C♯ type	SML.NET type
System.Boolean	bool	bool
System.Byte	byte	Word8.word
System.Char	char	char
System.Double	double	real
System.Single	float	Real32.real
System.Int32	int	int
System.Int64	long	Int64.int
System.Int16	short	Int16.int
System.SByte	sbyte	Int8.int
System.String	string	string
System.UInt16	ushort	Word16.word
System.UInt32	uint	word
System.UInt64	ulong	Word64.word
System.Exception	System.Exception	exn
System.Object	object	object

types, their C# equivalent and their CLR name. These tables show no overlap between the programming language type names and those used by the CLI. In the evaluation, we did experiment with making it easier to simplify type specification by allowing the use of truncated versions of CLI types names in which the namespace is removed. Hence, the use of `Int32` and `Void` in the XML of Figure 5.5. While these truncated versions are shorter to write, they make it easier to make mistakes. For example, in writing "`System.String`", we found the capitalization of `System` to be a reminder to capitalise the '`String`' portion. When the namespace was removed, it was easier to forget that the CLI type was being used, and so we reverted to using language-specific monikers. For example, '`string`', all lower case, was used instead of '`String`' with the capital first letter. These mistakes are hard to spot, since it appears that the type is correctly written. Generally, user types present less difficulty, as their name and namespace holds across language boundaries, but there are still quirks when user types are exported as nested classes. For instance, class types exported by SML.NET are nested in their respective module. A class `Logger` defined in module `ML_Logger` would be accessed using the moniker `Aspect_ML_Logger+Logger`. This moniker is used in the XML of Figure 5.6 in order to select the `Logger` nested class from an assembly written in SML.NET.

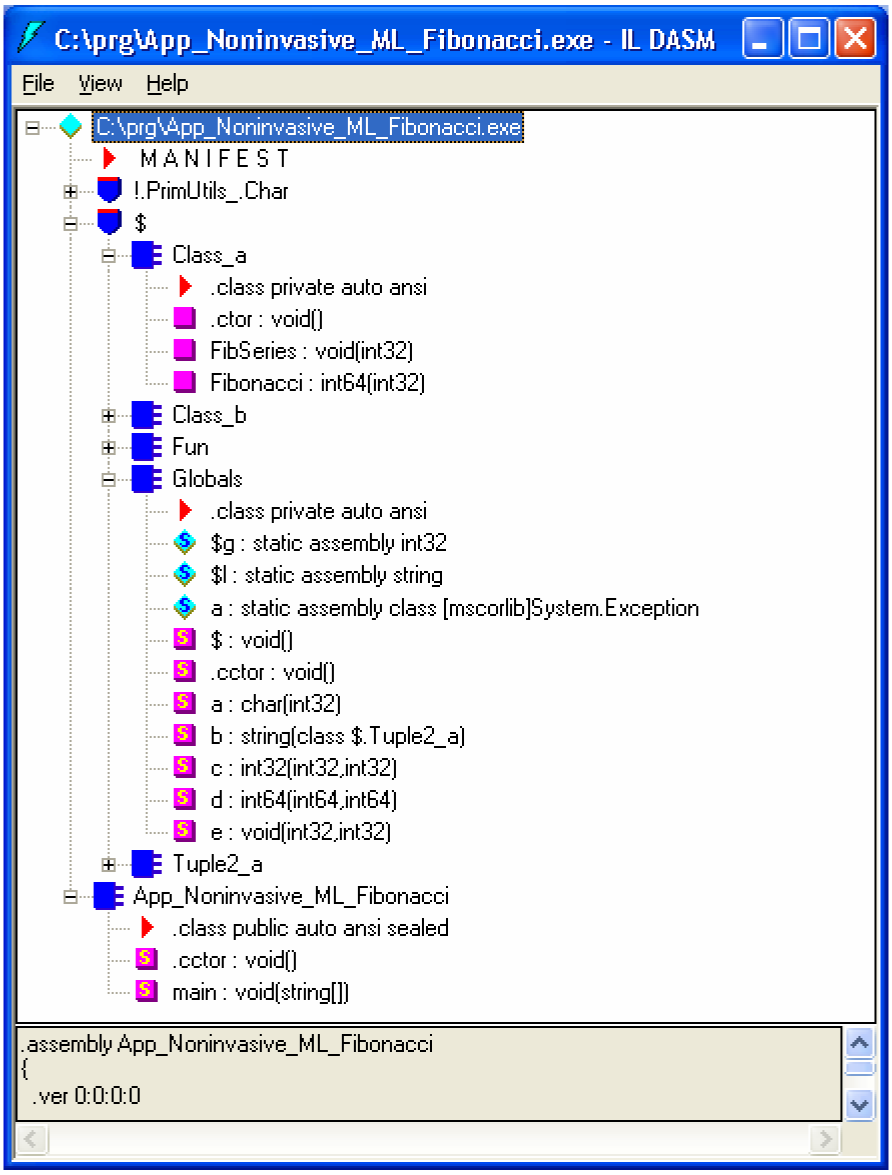
```
<?xml version="1.0" encoding="utf-8" ?>
  <ax:aspect xmlns:ax="http://aosd.dsg.cs.tcd.ie/XMLSchema"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://aosd.dsg.cs.tcd.ie/XMLSchema
             file:aspect_Schema.xsd">
  <name>LoggingAspect</name>
  <assembly>Aspect_ML_Logging</assembly>
  <type>Aspect_ML_Logging+Logger</type>
  <body>
     ...
  </body>
</ax:aspect>
```

Figure 5.6: Crosscutting specification for logging with SML.NET type implementing logging in bold.

```
structure App_Noninvasive_ML_Fibonacci
  : sig val main: string option array option -> unit
    end =
struct
_classtype FibonacciSeries()
 with
  Fibonacci (n) =
    case(n) of
      0 => (Int64.fromInt(1))
    | 1 => (Int64.fromInt(1))
    | n => (this.#Fibonacci(n-1) + this.#Fibonacci(n-2))
 and
  FibSeries (n) =
    case(n) of
      ~1 => ()
     | n => (this.#FibSeries (n-1);
             print "Element\t"; print (Int.toString (n)); print "\t value \t";
            print (Int64.toString(this.#Fibonacci (n))); print "\n" )
 end
 fun SelfTest (elements, times) =
  let
        val fibML = FibonacciSeries()
  in
        case(times) of
          0 => ()
        | n => (fibML.#FibSeries(elements); SelfTest(elements, times-1))
  end
 fun main  (a : string option array option) =
  let
    val elements = 10
    val times = 1
  in
    SelfTest(elements, times)
  end
end
```

Figure 5.7: SML.NET implementation of application to calculate Fibonacci Series elements.

Our evaluation also noted a severe problem with the accidental selection of join points when property-based crosscutting without attributes is used. Section 3.1.1.1 discussed two methods in which aspect-based properties supported property-based crosscutting. In the first instance, a pointcut designator's argument can be made more general by the use of regular expressions in the pointcut designator's argument. However, such regular expressions can make unexpected join point selections. Before evaluation, we made the general assumption that these extra join points could be spotted in source code. If this were the case, then with a careful examination of component source the crosscutting specification could be more finely crafted to remove the superfluous join points. However, evaluation tests involving components written in SML.NET indicate superfluous join points are not always visible from source. Assemblies generated by the SML.NET compiler can contain considerably more types than could be inferred from the source code. For example, Figure 5.7 defines a SML module with methods `main` and `SelfTest` at the module level and methods `Fibonacci` and `FibSeries` in the class `FibonacciSeries`. Using the directive "`export App_Noninvasive_ML_Fibonacci`" to compile this source results in an assembly containing a surprising number of additional types. As shown in Figure 5.8, an Ildasm-generated view of the type definitions uncovers a large number of types for which there are no explicit declarations in the source code. As expected, there is a type corresponding to the module that contains the implementation of `main` and `SelfTest`, and there is a class corresponding to the `FibonacciSeries` class declaration that contains the implementations of `Fibonacci` and `FibSeries`. The difficulty is that there are other types such as `Globals` with methods such as "`static char a(int32 A_0)`" that would match property-based crosscut for logging shown in Figure 5.3.



```
C:\prg\App_Noninvasive_ML_Fibonacci.exe
    MANIFEST
    !.PrimUtils_.Char
    $
        Class_a
            .class private auto ansi
            .ctor : void()
            FibSeries : void(int32)
            Fibonacci : int64(int32)
        Class_b
        Fun
        Globals
            .class private auto ansi
            $g : static assembly int32
            $l : static assembly string
            a : static assembly class [mscorlib]System.Exception
            $ : void()
            .cctor : void()
            a : char(int32)
            b : string(class $.Tuple2_a)
            c : int32(int32,int32)
            d : int64(int64,int64)
            e : void(int32,int32)
        Tuple2_a
    App_Noninvasive_ML_Fibonacci
        .class public auto ansi sealed
        .cctor : void()
        main : void(string[])
```

```
.assembly App_Noninvasive_ML_Fibonacci
{
  .ver 0:0:0:0
```

Figure 5.8: Ildasm view of types contained in assembly written in SML.NET source in Figure 5.7.

5.1.2 Reuse with Attribute-based Property Selection

To evaluate support for attribute-based property selection, we contrast the use of attribute types with the use of custom crosscutting and look for the attributes to be consistent with language-independence and to avoid the need to modify crosscutting specifications. To provide this contrast, the language-independence tests of section 5.1.1 are re-implemented using attribute-based property selection in section 5.1.2.1. So in section 5.1.2.1 it is as though aspect-component binding were conducted by an attributed component writer who annotated the component with attribute types. The attributed component writer role is described in section 3.2 of Chapter 3. Attributes are introduced into the source code for components written in C#, VB.NET, and SML.NET, as the CLS producer status of these code generation tools guarantees that attributes in source code will appear as metadata tags in the assembly that results from compilation. The advantages of using an attribute type can be seen in improved weaving performance in that the time required to weave a component is usually reduced. Another advantage is the simplicity of specifying aspect-component bindings. The slight performance improvements gained during composition when attribute-based property selection is used are likely due to simplification of join point matching.

In section 5.1.2.2, we note the qualitative advantages observed with the use of attribute-based property selection. Specifically, the use of attributes offers a more succinct and accurate means of applying crosscutting functionality; however, attributes prove difficult to use when call join points implemented by legacy components need to be manipulated.

5.1.2.1 Evaluating Attribute-based Property Selection

To review the discussion in section in 3.1.1, the crosscutting specifications implemented by the aspect-based property writer for an aspect-based property are meant to exploit attributes. These specifications are complemented with an attribute type that allows component source code to access the functionality of an aspect-based property. In contrast to custom crosscutting, aspect-based properties use attribute type names in place of join point implementation details such as types and type member signatures. When using attribute type names, the grammar for pointcut specifications is unchanged when it comes to the primitive pointcut designators available, but the parameters used for aspect-based properties vary. Rather than signature or type name arguments, primitive pointcut designators are parameterised with attribute tags describing the attribute type name. In the case of the CLI, these attributes are implemented by custom attribute types.

```
<execution>
  <method_signature>
    <return_type>
      <type_name>*</type_name>
    </return_type>
    <join_point_type>
      <type_name>*</type_name>
    </join_point_type>
    <method_name>*</method_name>
    <parameters>
      <parameter><type_name>Int32</type_name></parameter>
    </parameters>
  </method_signature>
</execution>
```

```
<execution>
  <attribute>Logging</attribute>
</execution>
```

Figure 5.9: Contrast between crosscutting semantics produced by an application integrator (top) and an aspect-based property writer (bottom).

The contrast between aspect-based property crosscuts and custom crosscuts can be seen in Figure 5.9. The top pane of the figure contains the execution pointcut specification used in Figure 5.3 to select execution join points for logging. In this pane, the selection of method execution join points is based on a method signature. In the bottom pane of Figure 5.9, the specification is revised to select methods tagged with an attribute type with the name `Logging`. This second version contains considerably fewer terms than the first. Of course, it is reliant on the ability to associate attributes with methods to be logged, and in our programming model this is done by annotating method source with an attribute type. Recall that attribute types provide an API for accessing aspect-based property functionality. For example, the attribute type in Figure 5.10 would form the basis of logging that is applied to method invocations only.

```
[AttributeUsage(AttributeTargets.Method)]
public class Logging : Attribute
{
  public Logging() {}
}
```

Figure 5.10: Implementation of attribute type for accessing logging provided by an aspect-based property.

An example application of attribute types is shown in Figure 5.11, where methods of the Fibonacci series algorithm in Figure 5.1 are bound to logging functionality. This example emphasises the attribute annotations by marking them in bold. In our CLI implementation, the attribute types are modelled as metadata extensions. This raises the possibility of modifying the metadata of an existing assembly as a means of annotating a component with attributes in the absence of source code. While this is theoretically possible, at present there are no tools to do so.

```
public class FibonacciSeries
{
  [Logging]
  public void FibSeries(int seriesLen) {
    for (int i = 0; i<= seriesLen; i++) {
      long result = Fibonacci(i);
      System.Console.WriteLine("Element \t"+ i+ "\tvalue \t"+result);
    }
  }

  [Logging]
  public long Fibonacci(int n) {
    if (n > 1)
      return this.Fibonacci(n-1) + this.Fibonacci(n-2);

    return 1;
  }
}
```

Figure 5.11: Fibonacci series enumerator annotated with attributes to identify methods for logging.

In our evaluation we are concerned with determining whether attribute-based property selection can be used in a language-independent manner and whether its use avoids the need to modify an existing aspect. To do so, we duplicated the language-independence tests of the previous section with crosscutting semantics revised to exploit attribute types for join point selection. Modification of the language-independent composition tests involves two steps. First, the crosscutting specification of the each logging aspect is revised to use an attribute for method selection. These changes are analogous to the changes made between the top and lower panes of Figure 5.9. Secondly, new versions of the components targeted for weaving are created in which attributes are applied in a similar fashion to Figure 5.11. Specifically, attribute types are applied to the `Fibonacci` and `FibSeries` methods, whose execution is to be logged. These new aspect-based properties are then composed with the new components, and average weave times are collected for comparison with the results of the previous weaving tests.

Our evaluation indicates that language-independence was preserved without introducing additional overhead to weaving. Language-independence is preserved in that the same attribute types were used to annotate components, regardless of the programming language used to implement the component. Assemblies resulting from the weaves passed the verification testing of Peverify, and weaving generally took less time than when custom crosscutting was used to specify aspect-component bindings. The weave times for attribute-based property selection are contrasted with their custom crosscutting equivalents in Table 5.4. The better performance with attribute-based property selection is likely due to a decrease in the number of XML tags that must be examined to determine a match between a method and an execution primitive pointcut designator. As is evident from

Figure 5.9, attribute-based specifications have considerably fewer XML elements. The reuse of aspect-based properties did not require the revising of the aspect-based property's crosscutting specification. This was also the case in our example of custom crosscutting; however, with attribute-based property selection the crosscutting specification did not dictate component implementation.

Table 5.4: Contrast of execution times for attribute-based and noninvasive selection of logging.

	Aspect Type Language		
Component Type Language	C#	VB.NET	SML.NET
C# (112 instructions)	143 ms (160 ms)	148 ms (169 ms)	153 ms (169 ms)
VB.NET (131 instructions)	92.5 ms (110 ms)	90.1 ms (106 ms)	93.3 ms (105 ms)
SML.NET (444 instructions)	121 ms (168 ms)	123 ms (165 ms)	123 ms (151 ms)

Invasive aspect results above, noninvasive below in parenthesis

ms – milliseconds or 10^{-3} seconds

Trials performed with debug build on 497MHz PentiumIII laptop with 384Meg of RAM under WindowsXP

5.1.2.2 Advantages and Disadvantages of Attribute-based Property Selection

Our evaluation noted the use of attribute-based property selection to be substantially simpler than custom crosscutting with the exception of call join points. Essentially, attribute-based property selection made writing aspect-component bindings less error prone, and in our trials only explicitly annotated types and type members were labelled with attributes in the compiled component.

Attribute-based property selection provides an alternative means of identifying CLI metadata that avoids mistakes made in a custom crosscutting specifications that are extremely difficult to detect. Recall that writing custom crosscutting involves specifying join points in terms of metadata that is native to the CLI. On the one hand, most programmers do not think in terms of CLI types, and so they are apt to make mistakes when they are forced to translate an API documenting a component in terms of language-specific types to the underlying CLI native types. On the other hand, there is little help available from the weaver for detecting erroneous type specifications, as it is hard to design a weaver that can distinguish between types that are specified correctly and those that are specified in error. For instance, the method parameters in Figure 5.11 are of type `int`. `int` is the C# moniker for the CLI type `System.Int32`, and thus the short form `Int32` appears in the method parameter specification of Figure 5.9. Should the type `int` appear accidentally, one would expect the weaver to complain. However, it is legitimate for a programmer to define a custom CLI type by the name of `int` in a different namespace. Even if we require that type names in the crosscutting specifications include a full namespace, `int` is still a valid user defined type. Attribute-based property selection avoids the issue of detecting errors made when the language type name is mapped to the CLI type name mappings, since the placement of attributes on types or type members avoids the need to deal with join point selection in terms of CLI-specific type names. In effect, the use of attributes represents the introduction of language-independent monikers for types and type members.

```
structure App_Invasive_ML_Fibonacci
  : sig val main: string option array option -> unit
    end =
struct
_classtype FibonacciSeries()
 with
  {Aspect_CS_Logging.Logging()} Fibonacci (n) =
    case(n) of
      0 => (Int64.fromInt(1))
    | 1 => (Int64.fromInt(1))
    | n => (this.#Fibonacci(n-1) + this.#Fibonacci(n-2))
 and
  {Aspect_CS_Logging.Logging()} FibSeries (n) =
    case(n) of
     ~1 => ()
    | n => (this.#FibSeries (n-1);
            print "Element\t"; print (Int.toString (n)); print "\t value \t";
            print (Int64.toString(this.#Fibonacci (n))); print "\n" )
 end

 ...

end
```

Figure 5.12: SML.NET implementation of Figure 5.7 updated to exploit custom attributes.

Also, attribute-based property selection has less difficulty with unexpected join point selection, since attributes follow the implementation of the tagged method. With revisions to include attributes, the SML-based Fibonacci series algorithm in Figure 5.7 takes on the appearance of that of Figure 5.12, where attributes appear in bold. Note that the definitions of `main` and `SelfTest` have been removed for brevity. As before, additional helper types will appear in the compiled assembly. However, an examination of the metadata of the assembly indicates that only those methods explicitly tagged at the source code level will have their metadata description annotated by the logging attribute in the compiled assembly. Thus, applying logging on the basis of attributes rather than method signature, constrains logging to the `Fibonacci` and `FibSeries` methods. VB.NET and C# showed similar behaviour in that only methods annotated in source code were annotated in the compiled assembly. However, these languages introduced no new types beyond those in source code and they introduced only a minor number of type members such as default constructors.

While updating the logging crosscutting specification, we noticed that attribute-based property selection was not as useful when it came to selecting call join points. Using attribute-based property selection, the signature argument of a call join point will be substituted with an attribute. This attribute must appear on the declaration of the method being called; however, in many cases the method being called is implemented by a legacy component. This is the case with methods from the CLI's Base Class Library such as the console output methods `Write` and `WriteLine`. Depending on the CLI implementation, source code may not be available, as is the case for Microsoft's .NET Framework implementation. Thus, in practice it is quite difficult to use attributes as the argument of a call primitive pointcut designator. A possible solution to this limitation is to allow call primitive pointcut designators to exploit parameterised attributes in which an attribute applied to a method can point out calls in the method's body. Such a solution is the subject of future research.

5.2 Aspect-Based and Context Crosscutting Functionality Compared

This section compares crosscutting concerns implemented with a contextual composition mechanism to those implemented with aspect-based properties. Our evaluation starts by examining the limits to context properties. This involves implementing crosscutting functionality in terms of context properties and contrasting this with equivalent functionality implemented as an aspect. These custom context properties are written using extensible contextual composition available with CLR contexts. This choice also avoids the need to discount platform differences in our performance comparisons, as both CLR contexts and our prototype weaver target the same platform. The crosscutting functionality chosen is high accuracy execution time profiling, which is available in the CLI as an API. Unfortunately, profiling is not sufficiently computationally intensive to ascertain the performance characteristics of context properties. The calculation of Fibonacci series algorithms according to the algorithm described in section 5.1 involves an exponentially growing number of method invocations, and so we examine the performance characteristics of logging applied via context properties to those of the aspect-based solution developed in the language-independence section. Finally, both technologies are applied to the task modifying an existing component. In this case, the task is to implement a memoization performance enhancement for the Fibonacci series algorithm.

5.2.1 Qualitative Comparison

In section 5.2.1.1, which follows, we use the task of implementing profiling to demonstrate the ability of aspect-based properties to match CLR context properties in terms of the functionality that can be implemented. Then, in section 5.1.1.2 we point out how the richer join point model available with aspect-based properties allows them to avoid preplanning issues with context properties and to provide better tailorability.

5.2.1.1 Profiling Implemented with CLR Contexts and Aspect-Based Properties

A problem that allows us to contrast contextual composition with aspect-based properties is that of profiling method invocations. The need for a tidy system for profiling method execution time is evident from the test results of the previous sections. Take for example Table 5.4. Here, 18 timing results were shown, and each corresponds to an average for three trials. Thus, at least 54 different executions had to be measured and their value stored. One option for capturing this data is to add timing code to the code that calls the weaver; however, we have to consider the number of locations at which code it updated in order to make this change. Two applications were used to generate the results in Table 5.4: one uses attribute-based property selection and one uses custom crosscutting. Each application is responsible for weaving a test group, which involves three different weaves, which means that code to report execution time has to be placed at six points throughout the application. We could refactor these invocations into a single method that provides profiling, but an attempt at such a refactoring would require retesting the new program design to verify that the semantics had not changed. Coincidentally, embedding timing code is the kind of tangling problem that is addressed by AOP. Thus, it should be possible to model profiling with an aspect-based property. Being a crosscutting functionality, it should also be possible to model profiling as a context property.

Using CLR contexts, the behaviour of the context property to handle execution time profiling becomes a message sink that accesses the CLI's profiling API. The CLI provides extensible context properties via message sinks, which intercept method calls received by an object and wrap them with additional functionality. At the core of a message sink class is a `SyncProcessMessage` method that is invoked whenever a synchronous method call is invoked on the object to which the message sink is attached. The `SyncProcessMessage` method is passed a reified version of the method invocation. The `SyncProcessMessage` can then perform processing and optionally pass on the invocation to the target object. The `SyncProcessMessage` method responsible for profiling method execution is shown in Figure 5.13. The method uses the WIN32 API `QueryPerformanceCounter` method to get high precision measurements of the execution time of a method, where the actual frequency of this clock is given by `QueryPerformanceFrequency`.

```
public IMessage SyncProcessMessage(IMessage msg)
{
  // We only want to process method calls
  if (!(msg is IMethodMessage))
    return m_next.SyncProcessMessage(msg);

  /// Record invocation details.
  RecordMethodDetails(msg);

  /// Start timer to measure weaving speed.
  ///
  IMethodMessage callMsg = msg as IMethodMessage;
  IMessage returnMethod = null;
  long ctr1 = 0, ctr2 = 0, freq = 0;

  if (QueryPerformanceCounter(ref ctr1)!=0) // <--------- Begin timing.
  {
    returnMethod = m_next.SyncProcessMessage(msg); // <-- Call profiled method

    QueryPerformanceCounter(ref ctr2);  // <-------------- Finish timing.
    QueryPerformanceFrequency(ref freq);

    RecordExecutionTime(ctr1, ctr2, freq);
  }
  else
  {
    this.sw.Write("High-resolution counter not supported.");
  }
  return returnMethod;
}
```

Figure 5.13: Message sink written in C# that is used to implement the functionality of an execution time profiling context property.

Message sinks implementing a context property are attached to context-bound objects during construction under the direction of a context attribute. The term "context-bound object" is CLI specific, and it refers to a type of object to which message sinks can be attached. Context-bound objects are those that directly or indirectly inherit from class `ContextBoundObject`. Context attributes are a subtype of custom attributes that bootstrap the process of attaching a message sink to a newly instantiated object. When a context-bound object is instantiated, associated context attributes are instantiated and asked to add themselves to the list of context properties applied to the context-bound object. These properties are in turn responsible for assigning message sinks to the object, and these message sinks implement context properties. As a concrete example, Figure 5.14 provides the definition of a class of context attribute that attaches an execution time profiling property to a context-bound object. The implementation of the property being attached follows in Figure 5.15.

```
[AttributeUsage(AttributeTargets.Class)]
public class ProfileExecutionTimeAttribute : ContextAttribute
{
  public ProfileExecutionTimeAttribute() :
           base("ProfileExecutionTimeAttribute") {}

  public override void GetPropertiesForNewContext(IConstructionCallMessage ccm)
  {
    ProfileExecutionTimeProperty newProp = new ProfileExecutionTimeProperty();
    ccm.ContextProperties.Add(newProp);
  }
}
```

Figure 5.14: Context attribute to bind a profiling property to a context-bound object.

```
public class ProfileExecutionTimeProperty : IContextProperty,
                                            IContributeObjectSink
{
  public IMessageSink GetObjectSink(MarshalByRefObject o, IMessageSink next) {
    return new ProfileExecutionTime(next);
  }

  public bool IsNewContextOK( Context newCtx ) { return true ; }
  public void Freeze(Context newContext) { }
  public string Name { get { return "ProfileExecutionTimeProperty"; } }
}
```

Figure 5.15: A context property that implements profiling using the message sink of Figure 5.13.

Unfortunately, it is not possible to apply context property profiling to calls to weaving API methods. As mentioned, the CLR context properties only work with objects that are subtypes of `ContextBoundObject`, and only incoming messages can be captured. Thus, the profiling context property would have to be applied to the weaver API, rather than invocations by clients of the weaver API. To do so, the weaving library would have to be updated to sit in a subtype of `ContextBoundObject`, and the methods in the API would have to be changed from static methods to instance methods. Such modifications are far reaching in that they would break significant portions of existing test code and require recompilation of the weaver. A reasonable alternative is to leave the weaver 'as is' and access the weaving API via a wrapper interface along the lines of that pictured in Figure 5.16. In this figure, the type `ProfilingWrappers` inherits from `ContextBoundObject` and `ProfilingWrappers` is tagged with the `ProfileExecutionTime` context based property, also shown in bold. The class allows profiling through the `WeaveCallWrapper` method that forwards calls to the weaving API. As an instance method in a type inheriting from

ContextBoundObject, WeaveCallWrapper is able to be influenced by the context property.

```
[ProfileExecutionTime()]
public class ProfilingWrappers : ContextBoundObject
{
  public TimingCallWrappers() { }

  public void WeaverCallWrapper(
      string component, string componentPath,
      string aspect, string aspectPath)
  {
    TCD.CS.DSG.WeaveDotNet.Weave(component, componentPath, aspect, aspectPath);
  }
}
```

Figure 5.16: Application of context profiling to weaver profiling via a wrapper class.

Rather than a message sink, the aspect-based profiling property uses around advice. This around advice, shown in Figure 5.17, uses a `Proceed` invocation to pass control to the join point after starting the timer. After the join point returns, the timer is stopped. Unlike a message sink, the advice return type matches that of the method that it was replacing, whereas message sinks return a reified result. The profiling aspect-based property's crosscutting specification is written in terms of attributes, and the specification is shown in Figure 5.18. The `TimeCalls` named pointcut that organises join point selection binds profiling to each individual method invocation for methods tagged with an attribute of type `ProfileAllCalls`. `TimeCalls` combines a `withincode` primitive pointcut designator that selects all join points in a tagged method with a `call` designator that limits this set of join points to method invocations. The advice semantics of Figure 5.19 further limit the set of methods being profiled by selecting only those with a `void` return type.

```
public void ProfileExecution()
{
  RecordMethodDetails();

  long ctr1 = 0, ctr2 = 0, freq = 0;
  if (QueryPerformanceCounter(ref ctr1)!=0) // <--------- Begin timing.
  {
    object[] temp = new object[0];
    ((IAspect)this).Proceed(temp); // <------------------ Call profiled method

    QueryPerformanceCounter(ref ctr2); // <-------------- Finish timing
    QueryPerformanceFrequency(ref freq);

    RecordExecutionTime(ctr1, ctr2, freq);
  }
  else
  {
    this.sw.Write("High-resolution counter not supported.");
  }
}
```

Figure 5.17: Method implementing behaviour of advice execution time profiling.

```
<item>
  <named_pointcut>
    <modifier><public/></modifier>
    <name>TimeCalls</name>
    <pointcut>
      <and>
        <pointcut>
          <primitive>
            <withincode>
              <attribute>ProfileAllCalls</attribute>
            </withincode>
          </primitive>
        </pointcut>
        <pointcut>
          <primitive>
            <call>
              <method_signature>
                <return_type><type_name>*</type_name></return_type>
                <join_point_type><type_name>*</type_name></join_point_type>
                <method_name>*</method_name>
                <parameters><param_wildcard/></parameters>
              </method_signature>
            </call>
          </primitive>
        </pointcut>
      </and>
    </pointcut>
  </named_pointcut>
</item>
```

Figure 5.18: Pointcut to associate method call profiling with the `ProfileAllCalls` attribute.

```
<item>
  <advice>
    <around>
      <return_type>Void</return_type>
      <pointcut>
        <primitive>
          <pointcutId><name>TimeCalls</name></pointcutId>
        </primitive>
      </pointcut>
      <behaviour><name>ProfileExecution</name></behaviour>
    </around>
  </advice>
</item>
```

Figure 5.19: Advice to associate join points in the `TimeCalls` with around advice of Figure 5.17.

Despite having differences in implementation, the context-based profiling matches the aspect-based profiler in terms of the functionality. The results generated by profiling weaving execution time using a context property and aspect-based profiling are shown in Table 5.5. The tests performed were those involving attribute-based property selection described in section 5.1.2. The identical nature of aspect-based property profiling figures here and those in Table 5.4 reflect the fact that aspect-based profiling was used to gather data for section 5.1. The execution times measured by context-based properties are typically longer than their aspect-based property equivalent. This is likely in part due to execution overheads inherent to context-bound objects, which we investigate further in the performance comparison.

Table 5.5: **Profiling using context property with aspect-based property results below in parentheses.**

Component Type Language	Aspect Type Language: C#	Aspect Type Language: VB.NET	Aspect Type Language: SML.NET
C# (112 instructions)	143 ms (143 ms)	150 ms (148 ms)	154 ms (153 ms)
VB.NET (131 instructions)	100 ms (92.5 ms)	98.1 ms (90.1 ms)	101 ms (93.3 ms)
SML.NET (444 instructions)	128 ms (121 ms)	120 ms (123 ms)	129 ms (123 ms)

Results measured by context-based properties above, aspect-based versions below in parenthesis, ms – milliseconds or 10-3 seconds

Trials performed with debug build on 497MHz PentiumIII laptop with 384Meg of RAM under WindowsXP

5.2.1.2 Advantages of Aspect-Based Properties over Context Properties

In our evaluation, we noted specific instances in which component architecture changes required to access CLR context properties introduced preplanning problems and limit support for tailorability. First, the component instance inheritance hierarchy is modified as component instances must inherit from the `ContextBoundObject` class. Such changes constitute preplanning requirements that we wished to address with aspect-based properties. Second, the means by which the component instance interacts with other objects in the system is changed, as method invocations will always be reified in order to be processed by the message sinks that a context property attaches to the component instance. As highlighted in Figure 5.16, context-based profiling requires the code being profiled to be crafted around profiling. This requirement constrains the tailorability available with CLR contexts as the scope of component functionality that can be influenced by the context properties is limited.

The problems with CLR contexts described above are addressed with the richer join point model available with aspect-based properties. Whereas aspect-based properties offer call, execution and field access join point selection, extensible contextual composition only supports the manipulation of method execution join points with what amounts to around advice. Moreover, the set of method execution join points available is quite constrained.

Static calls, i.e. those not made on class instances, cannot be influenced by context properties. Only invocations crossing object boundaries can be considered, and so methods called on an object instance by that instance cannot be influenced either. As well as having a more limited set of join points to influence, context properties lack abstractions to refine this set. For instance, the profiling context property had to be applied to all method invocations on a tagged type, whereas an aspect-based property could narrow the set of profiled calls to those made by a specific method. Thus, the architectural changes required by CLR contexts are principally due to the narrower set of join points available for manipulation with context properties.

5.2.2 Overhead Performance Comparison

Our performance tests indicate the execution overhead of aspect-based properties to be an order of magnitude lower than that of context properties. In a comparison of logging implemented as a context property and logging implemented as an aspect-based property, the aspect-based properties executed ten times faster. Subsequent research indicated that use of the CLI's reflective API when gathering logging data introduces significant execution overhead, so a second comparison was done using context properties and aspect-based properties that had all internal functionality removed. This second test indicates the performance of aspect-based properties to be better than expected, but the performance degrades as the number of parameters passed to advice increases.

In the following sections we discuss the testing procedure and subsequent results.

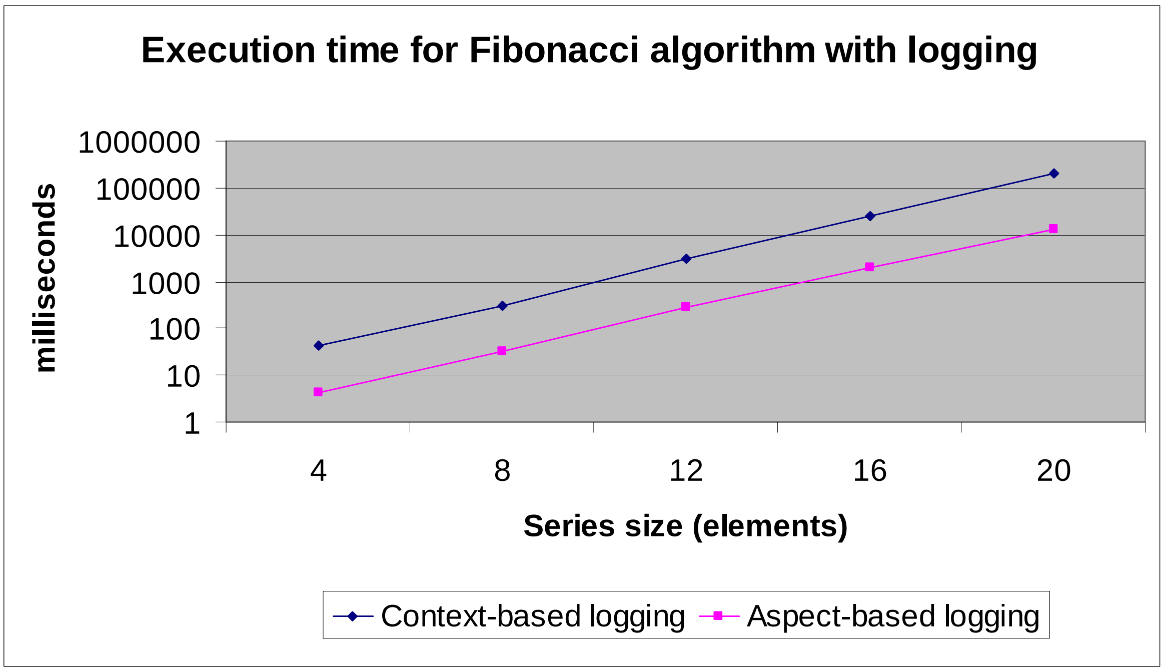
5.2.2.1 Comparison of Logging Functionality

Our initial performance tests involved applying logging to a modified version of the Fibonacci algorithm. As CLR contexts can only influence inter-object method invocations, the recursion of Figure 5.11 had to be rewritten in terms of mutually recursive invocations made by two different objects so that method executions of interest were exposed as external invocations. An implementation that satisfies this requirement with a minimum of object instantiations is shown in Figure 5.20. The algorithm uses mutual recursion between objects, and for clarity these objects are of different types. Logging is applied with attribute-based property selection for both context properties and aspect-based properties. In the case of context-based logging, the two types involved in the algorithm are annotated with context attributes. In the case of aspect-based property logging, the two methods that are specifically of interest are annotated. Note that while inheriting from class `ContextBoundObject` and using mutual recursion is only required for the context property logging, for consistency the same algorithm implementation is used when aspect-based logging is examined.

```
public class FibonacciSeries : ContextBoundObject
{
  public static FibonacciSeries singleton;
  protected FibonacciSeries() {}
  static FibonacciSeries() { singleton = new FibonacciSeries(); }

  public void FibSeries(int seriesLen) {
    for (int i = 0; i<= seriesLen; i++)  {
      long result = FibonacciCalc.singleton.Fibonacci(i);
      System.Console.WriteLine("Element \t"+ i+ "\tvalue \t"+result);
    }
  }

  public long Fibonacci(int n) {
    if (n > 1)
      return  FibonacciCalc.singleton.Fibonacci(n-1) +
              FibonacciCalc.singleton.Fibonacci(n-2);
      return 1;
  }
}

public class FibonacciCalc : ContextBoundObject {
  public static FibonacciCalc singleton;
  protected FibonacciCalc() {}
  static FibonacciCalc() { singleton = new FibonacciCalc(); }

  public long Fibonacci(int n) {
    if (n > 1)
      return  FibonacciSeries.singleton.Fibonacci(n-1) +
              FibonacciSeries.singleton.Fibonacci(n-2);
    return 1;
  }
}
```

Figure 5.20: Fibonacci algorithm implemented as mutually recursive singleton objects that is used for performance comparisons.

Figure 5.21: Execution times for algorithm of Figure 5.20 with context and aspect-based logging.



Comparing the execution time of the logging implemented with each mechanism indicates a significant speed up in the case of aspect-based logging. During our trials, we noted that adding logging to the algorithm of Figure 5.16 slows the calculation of Fibonacci elements significantly, even when logging output is piped to a file. Of more interest is the difference in execution time between each approach to logging. Despite using the same base algorithm for both aspect-based and context logging, the context logging executed an order of magnitude slower than aspect-based logging. This difference in execution speed is captured in Figure 5.21, where the use of context-based logging increases execution time approximately ten fold over aspect-based logging in all cases. Moreover, the execution speed of context-based logging and aspect-based logging show no sign of converging as the number of Fibonacci elements to be calculated increases.

5.2.2.2 Comparison of Intercession Mechanism

The second performance evaluation focuses on the execution overhead of the intercession mechanisms of context properties and aspect-based properties. Our initial performance tests revealed significant overhead was incurred when the logging functionality made use of the CLI's reflection API. To determine if this overhead distorted the performance comparison, we devised a new test that examined the overhead of the intercession mechanism used by context properties and aspect-based properties. To begin, seven functions were written that do not contain any functionality. These methods are implemented in class `BlanksMethods` shown in Figure 5.22. Next, a context property and an aspect-based property were written to intercede at the start and end of each method execution join point, and to expose the join point's arguments and return value as typed formal parameters. Despite having access to these parameters, the context property and aspect-based property did not include any functionality, i.e., the before and after returning advice methods contained no behaviour.

```
public class BlanksMethods : ContextBoundObject
{
  public static BlanksMethods singleton;
  public BlanksMethods() {}
  static BlanksMethods() { singleton = new BlanksMethods(); }

  public void    BVoidVoid() { }

  public void    BVoidInt(int a) { }
  public void    BVoidIntInt(int a, int b) { }
  public int     BIntIntInt(int a, int b) {return a; }

  public void    BVoidObj(int a) { }
  public void    BVoidObjObj(Object a, Object b) { }
  public Object  BObjObjObj(Object a, Object b) {return a; }
}
```

Figure 5.22: Empty methods that will be influenced by aspect-based properties and context properties.

Performance profiling confirmed the intercession mechanism of aspect-based properties to be an order of magnitude faster than that of context properties, but this improvement degrades as the amount of join point execution context exposed by an aspect-based property increases. As with all performance tests in this section, the execution overhead was profiled using a debug build running on a 497MHz Pentium III laptop with 384Meg of RAM under WindowsXP Professional, and execution times were calculated using the laptop's high speed timer which operates at approximately 3.6 MHz. Our trials examined average execution overhead due to intercessions as the number of invocations increased from 10000 to 100000 in steps of 10000. We observed average execution overhead to increase linearly, and so values at 10000 and 100000 invocations were used as endpoints of a line from which the average overheads shown in Table 5.6 were calculated. The table indicates that the execution overhead of aspect-based properties is heavily dependent on the number of parameters that must be loaded onto the stack during advice invocation; however, in our trials aspect-based properties offer an even better performance increase than was alluded to in Figure 5.21.

Table 5.6: Execution overhead of intercession mechanism of context properties and aspect-based properties when applied to methods of Figure 5.22.

Target Method	Context Property Intercession Overhead (microseconds)	Aspect-Based Property Intercession Overhead (microseconds)
`Void BVoidVoid()`	50.27	1.521
`Void BVoidInt(Int32)`	53.72	3.202
`Void BVoidIntInt(Int32, Int32)`	55.08	3.711
`Int32 BIntIntInt(Int32, Int32)`	55.80	4.121
`Void BVoidObject(Object)`	53.96	2.725
`Void  BVoidObjObj(Object, Object)`	55.29	3.536
`Object BObjObjObj(Object, Object)`	55.43	3.665

5.2.3 Improving Legacy Component Performance

In this final phase of the evaluation, we apply a light-weight performance-enhancement to demonstrate aspect-based properties to be more useful for legacy components than context properties. The comparison tests so far have had the opportunity to revise component implementation to suit the crosscutting functionality being applied. In this section, we restrict the crosscutting concerns to working with a component as-is, i.e., without the ability to recompile the component. In contrast, the initial language-independence tests were able to synchronize the implementation of components to suit a custom crosscutting specification, and the second batch of language-independence tests were able to add attributes to suit an aspect-based property or context property. The programming task in this section is to implement and apply memoization for the purposes of improving component execution performance. Memoization has been used in the past to demonstrate the ability of aspects to improve existing code [Men'97], and it consists of caching a function's results for a given input to avoid having to recalculate the results. Memoization is well suited to functional programming languages such as SML.NET where functions are stateless and function execution is side-effect free. The target for memoization is the `Fibonacci` method of the SML.NET-based Fibonacci series enumeration algorithm shown earlier in Figure 5.12.

```
public class CacheResults : Aspect
{
  public CacheResults() { }

  const int cacheSize = 5;
  static long[] results = new long[cacheSize];
  static int[] inputs = new int[cacheSize];
  static int lastIndex = 0;

  static CacheResults() {
    for ( int i = 0; i<cacheSize; i++)
      CacheResults.inputs[i] = -1;
  }

  public long CacheInt_Int(int input)
  {
    for ( int i = 0; i<cacheSize; i++) {
      if ( input == inputs[i])
        return results[i];
    }

    object newResult = ((IAspect)this).Proceed(input);

    inputs[lastIndex]  = input;
    results[lastIndex] = (long)newResult;
    lastIndex = (lastIndex + 1)% lastIndex;

    return (long)newResult;
  }
}
```

Figure 5.23: Implementation of memoization aspect behaviour written in C#.

This test highlights the advantages of aspect-based properties, as it is not possible to implement the desired functionality with a CLR context property. The SML.NET algorithm does not inherit from `ContextBoundObject`, and so context properties are not applicable. However, it is possible to add memoization to an existing component using custom crosscutting, and it is possible to observe a significant improvement in execution time when doing so. Unlike the aspect discussed in [Men'97], our memoization aspect-based property is customised for the Fibonacci series enumerator and includes an aspect written in a different programming language. The implementation of aspect-based memoization is shown in Figure 5.23. The use of C# reflects the familiarity of the author with OO languages. In this context, the use of C# is consistent with the intent of language-independent AOP supported by aspect-based properties, which is meant to make it easier to write and apply aspects using existing developer knowledge. The algorithm implemented is not a good general purpose implementation of memoization. The algorithm is not guaranteed to work if a `-1` is used as one of the initial inputs. Fortunately, our SML.NET algorithm does not face such inputs. The algorithm is applied to join points in the `FibonacciCalc` pointcut of Figure 5.24 by the advice in Figure 5.25.

```
<item>
  <named_pointcut>
    <modifier><public/></modifier>
    <name>FibonacciCalc</name>
    <local_var_ref>
      <var_type>Int32</var_type>
      <var_name>data</var_name>
    </local_var_ref>
    <pointcut>
      <and>
        <pointcut>
          <primitive>
            <execution>
              <method_signature>
                <return_type><type_name>*</type_name></return_type>
                <join_point_type><type_name>*</type_name></join_point_type>
                <method_name>Fibonacci</method_name>
                <parameters><param_wildcard></param_wildcard></parameters>
              </method_signature>
            </execution>
          </primitive>
        </pointcut>
        <pointcut>
          <primitive>
            <args>
              <parameter>
                <formal_parameter_name>data</formal_parameter_name>
              </parameter>
            </args>
          </primitive>
        </pointcut>
      </and>
    </pointcut>
  </named_pointcut>
</item>
```

Figure 5.24: Custom pointcut to select Fibonacci method for memoization and expose input argument.

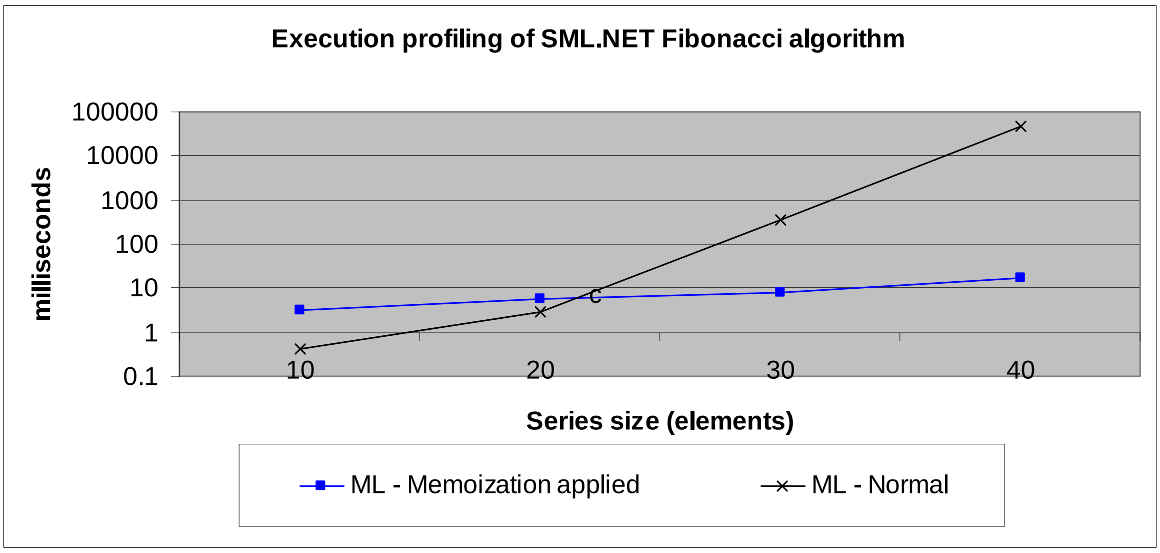
```
<item>
  <advice>
    <around>
      <return_type>Int64</return_type>
      <formal_param>
        <var_type>System.Int32</var_type>
        <var_name>input</var_name>
      </formal_param>
      <pointcut>
        <primitive>
          <pointcutId><name>FibonacciCalc</name></pointcutId>
        </primitive>
      </pointcut>
      <behaviour><name>CacheInt_Int</name></behaviour>
    </around>
  </advice>
</item>
```

Figure 5.25: Advice statement to apply memoization to pointcut specified in Figure 5.25.

When applied to the calculation of Fibonacci series elements, the memoization aspect-based property greatly improves performance. The results in Table 5.7 represent an average of three trials in which Fibonacci series of differing sizes were calculated on a

PentiumIII laptop running at 497MHz with 384Megs of RAM under WindowsXP Professional. The graph scale is logarithmic to highlight that the performance difference between memoized and non-memoized results for a lower number of elements is insignificant as compared to the benefits achieved when larger numbers of elements are calculated

Table 5.7: execution time of Fibonacci series calculations with and without a memoization aspect.



5.3 Usability Issues with the Programming Model

In addition to the concrete analysis of the previous section, anecdotal evidence from users has exposed some problems with the programming model of aspect-based properties as implemented by Weave.NET and in particular with the use of XML to specify crosscutting semantics and with the code generation support in the weaver. From one point of view, XML-based expressions are not necessarily language-independent, since they may be viewed as a language with the validating schema as its grammar. We address this criticism by pointing out that pointcut-advice operations would otherwise be language extensions. By expressing them in XML, these language extensions become consistent for all programming languages and thus the syntax is language-independent. Another complaint from users of Weave.NET is that the XML specifications are difficult to write and XML specifications are not succinct. Indeed, in section 5.1.1, we pointed out the ease by which property-based crosscutting can inadvertently select unwanted join points, and we explained the difficulty in verifying type specifications in signatures. Our solution here was to emphasis the use of attribute-based property selection to make aspect-component bindings explicit. However, there may be room for improvement by introducing visual tools such as those available with AMT [Han'01] or AJDT [Cle'04] to provide users with a view of affected code or tools such as LOOM.NET [Sch'03] to help in selecting join points for annotation by attributes or inclusion in custom crosscutting. As pointed out in Chapter 3, the XML grammar probably serves well as a backend format for GUI-based or language-based specification of crosscutting semantics than something to be written freehand.

In terms of implementation, and not design, the prototype weaver has difficulties in terms of completeness and its ability to weave components that are dependent on each other. A difficulty noted during testing was that the code generation architecture was not 100% complete. Due to the lack of programmer resources, some CLI elements such as delegate types are not emitted during weaving. Less obvious are the problems with emitting nested type constructors that have no body. The final problem was that of inadvertent assembly loading. As noted in section 5.1.1, the application of weaving loads an assembly, and it is impossible to unload that assembly. Furthermore, Weave.NET also loads all the types upon which an assembly being woven is dependent, and this loading prevents the assemblies of these types from being subsequently woven. This problem was dealt with in this evaluation by using reflection to avoid direct dependencies between assemblies that had to be woven during testing. The CLI’s reflection API was used to manually load assemblies and invoke required methods. While this works fine, it can lead to compilation errors. When the explicit dependencies are removed, our build environment could not determine automatically which assemblies needed to be updated during compilation. Moreover, the use of the reflective API obfuscates the resulting code.

5.4 Summary

In this chapter, we examined the ability of our prototype weaver to support the adoption of the aspect-based property programming model and we compared aspect-based functionality with context properties in terms of available functionality and performance. Our evaluation also noted usability problems that made writing crosscutting in XML error prone, and we noted accidental component loading to cause problems when our prototype weaver is used.

In terms of adoption, language-independent AOP allowed aspect-based properties to preserve existing developer skills, development tools and source code based, and attribute-based property selection avoided the need to learn how PA mechanisms worked in order to use them. In terms of language-independence, the prototype was shown to weave aspect-based properties, written in object-oriented, functional and procedural programming languages, with components also written in these programming languages. Attribute-based property selection was consistent with language-independence in that attribute types could be created in a variety of languages, and those created could be applied to any of the languages used for component development. Moreover, attribute-based properties avoided the need to write custom crosscutting to bind logging and profiling to components, hence their use required no learning on the part of a component writer. Also, we noted that property-based crosscutting in terms of implementation details does not well work in a multi-language environment due to the appearance of unexpected helper methods in the component; however, attribute-based property selection avoided this problem as attributes followed the method to which they were applied.

A comparison of aspect-based properties and context properties available for the CLI noted aspect-based properties to offer richer functionality, but to incur an order of magnitude less overhead. In comparison to the programming model for extending context-based properties for the CLI, aspect-based properties improved tailorability by offering a richer join point model, and they avoided preplanning issues by not requiring object instances or specifying inheritance requirements. Also, aspect-based properties support legacy components. More importantly, underlying intercession mechanism used by aspect-based properties was shown to operate ten to thirty times faster than that of context properties depending the number of typed formal parameters accessed by advice. This speed up was also observed when a logging aspect-based property was compared to a context property with identical functionality.

In terms of usability, the XML does provide a language-independent syntax for writing crosscutting in that it avoids the need to include languages extensions to specify crosscutting specifications, but work is needed to make XML easier to exploit. Currently, the code generation mechanism loads methods upon which a component is dependent. Our evaluation used the reflection API is used to remove dependencies between components, but in future the code generation mechanism should be upgraded.

Chapter 6 Conclusion

“Where’s the contribution?”
–Prof Vinny Cahill

This thesis is motivated by an interest in applying aspect-oriented programming (AOP) to addressing crosscutting functionality in software components. Our starting point was the use of contextual composition in component frameworks that support declarative selection of component framework services in which services are bound to component instances using method intercession. However, such contextual composition suffers from the lack of tailorability problem as well as the preplanning problem. To solve these problems, we introduced aspect-based properties. Aspect-based properties allow crosscutting functionality to be written with a pointcut-advice aspect-oriented mechanism that is bound to components with a load-time weaver. The difficulty with aspect-oriented programming is that it fails to address reusability and it introduces language-dependencies to composition. Aspect-based properties are able to overcome these problems with attribute-based property selection and language-independent AOP.

To conclude this thesis, we provide an overview of the thesis that identifies the contribution of each chapter. We then summarise how the goals set out for this thesis were achieved by the programming model formulated for aspect-based properties. Finally, we identify future implementation and research work.

6.1 Thesis Overview

In Chapter 1, we introduced contextual composition in the context of component frameworks. We pointed out that the difficulty with systems that model crosscutting concerns as context properties is that these systems suffer from the lack of tailorability problem in that the set of context properties is difficult to extend. Context properties also suffer the preplanning problem in which their use dictates component architecture requirements. Aspect-based properties were introduced as an AOP-based solution to these problems, which also avoided the reusability problems and language dependencies that AOP introduces.

In Chapter 2, we provided a survey of AOP that focused on five canonical aspect-oriented mechanisms, which were as follows:

- The pointcut-advice (PA) mechanism exemplified by the work of the AspectJ Team [Asp'00]
- Class composition exemplified by the work of the MDSOC Project [IBM'00a]
- Object-graph traversal exemplified by the work of the Demeter Project [Lie'00]
- Open class composition, which originated with mixins [Moo'86], but is exemplified by the inter-type declaration semantics of AspectJ
- Composition Filters (CF) object model extensions available with tools such as ComposeJ [Car'01]

These mechanisms were characterised in terms of their aspect model, which consists of a join point model, a means of identifying join points, and a means of modifying the semantics of join points. Our analysis concluded that both the CF and PA mechanisms are well suited to producing the same behavioural changes to join points as a contextual composition mechanism; however, the PA aspect model was easier to conceptualise, offered finer grained crosscutting and promised better performance. Unfortunately, AOP introduced language dependencies into aspect-component composition and reuse involved revising the aspect’s crosscutting specifications. Finally, we noted that load-time weaving met deployment requirements of components and provided support for clear-box crosscutting available with PA mechanisms.

In Chapter 3, we introduced a programming model in which crosscutting concerns were modelled with aspect-based properties. Aspect-based properties implement crosscutting concerns in a programming model characterised by:

- The availability of a PA mechanism
- Language-independent AOP
- Attribute-based property selection
- Load-time weaving

We provided an overview of application programming involving aspect-based properties, and we identified four independent programming roles and their products. The roles, their relationships and their products are summarised in Figure 6.1. These roles include the aspect-based property writer, who produces aspect-based properties corresponding to crosscutting functionality. The aspect model supports a pointcut-advice mechanism, in which pointcuts are written in terms of attributes annotated to component code. While the aspect-based property implements crosscutting functionality, it is its corresponding attribute types that provide an API for this functionality. The binding between attribute-based properties and components is written in terms of attribute annotations to component source by the component writer. Since attribute-based property selection makes no

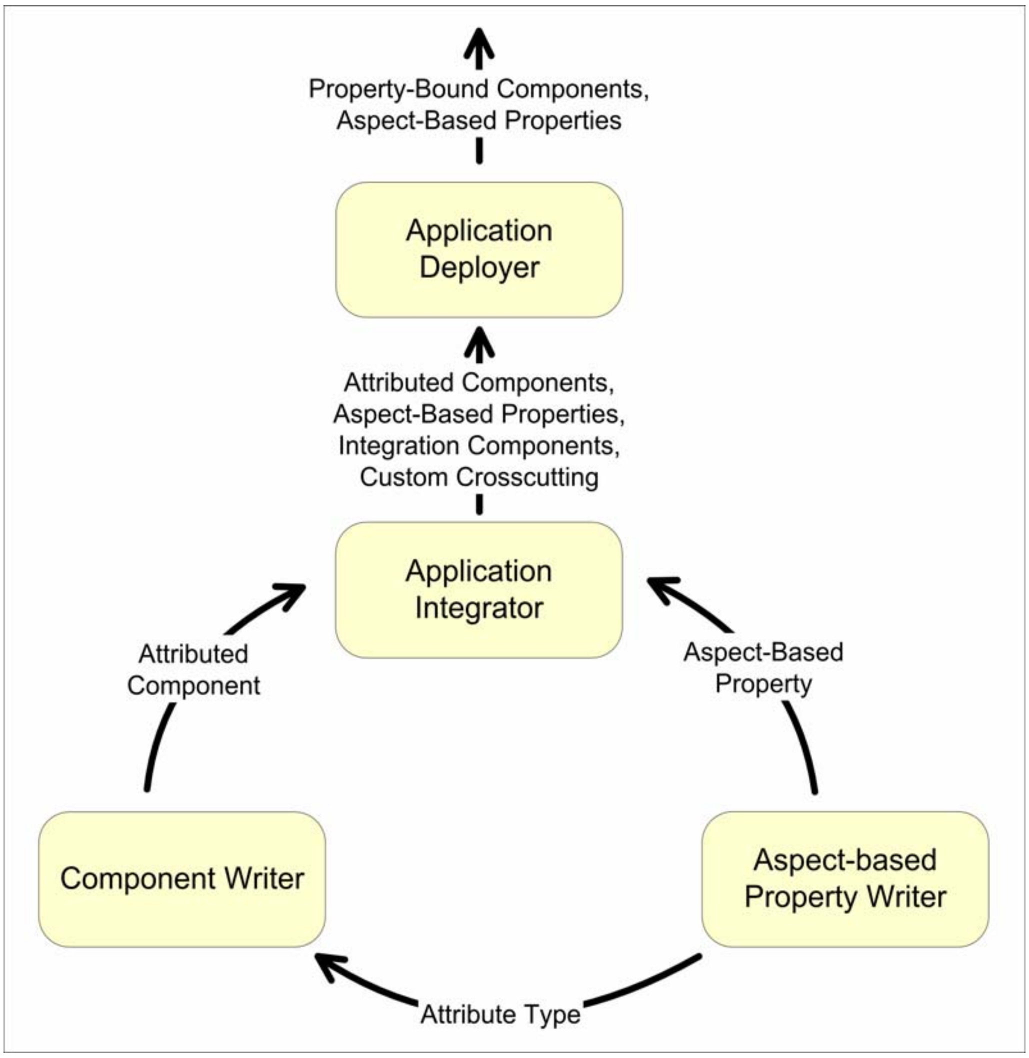


Figure 6.1: Overview of roles and products of development involving aspect-based properties.

changes to aspect-based properties, aspect-based properties can be reused without modification. Annotated components, referred to as attributed components, and aspect-based properties are selected by an application integrator. Aspect-based properties are meant to be used as-is, but the application integrator may also write custom crosscutting to accommodate legacy components. The application deployer is responsible for including a load-time weaver in the execution environment and for providing access to aspect-based properties and components to this weaver. Components that have been inspected at load-time for join points matching the pointcut specifications of aspect-based properties are referred to as Property Bound Components. The PA mechanism is based on that of AspectJ V1.0.6. PA specifications are expressed in XML to avoid introducing language extensions that would curb language-independence. Separate compilation of aspect-based properties and attributed components dictates limits to the PA mechanism. Types defined by attributed components are not available as typed formal parameters in the aspect-based properties. Keywords are available as methods inherited from a weaver-supplied type. Finally, pointcuts are expressed in terms of component metadata instead of language syntax.

In Chapter 4, we detailed the design of a prototype weaver that supports aspect-based properties for the Common Language Infrastructure (CLI). The CLI was chosen as the experimental platform for this weaver due to its explicit support for multilingual development and attribute annotation of component source. A schema was required for the XML to be processed by the weaver, and so rules that mapped BNF to XML were drawn up to convert the AspectJ grammar to XML schema. Composition and aspect behaviour was supported by a weaving library. We presented a design for this library that explained how it implemented two APIs: one for aspect-component weaving and one for AspectJ keyword emulation. Finally, we described the mechanism by which weaving was attached to an execution environment. Essentially, the execution environment was launched by a program that first wove application components with the aspect-based properties included in the component before passing control to the entry point of the application.

In Chapter 5, we used the prototype weaver to examine the programming model for aspect-based properties in terms of adoptability, and we compared this programming model to that of CLR contexts, which allow the set of context properties available for the CLI to be extended. The evaluation illustrated the language-independence and attribute-based property selection characteristics of aspect-based properties that smooth AOP adoption. In comparison to CLR context properties, aspect-based properties improved tailorability by offering a richer join point model. Aspect-based properties avoided preplanning issues by not imposing inheritance constraints and object-oriented characteristics, such as in instance methods, on types implemented by components. Finally, we noted that the overhead of aspect-based properties was an order of magnitude lower than context properties.

6.2 Thesis Contribution

We are especially interested in observations made during the evaluation of Chapter 5 that led us to conclude that aspect-based properties offered to replace contextual composition in component frameworks. With respect to existing contextual composition, we wanted to solve the tailorability and preplanning problems. These problems were observed to be solved in that aspect-based properties offered a richer join point model than CLR contexts and a programming model in which the weaver handled composition of components and crosscutting concerns instead of the component architecture. The richer join point model of attribute-based properties allowed memoization to be applied to an algorithm for calculating Fibonacci Series elements. This memoization optimization could not be recreated with CLR contexts, as they could not intercede in intra-object method invocations. In terms of preplanning, we observed that component instances wishing to take advantage of simple context properties such as logging and profiling had to make significant concessions in terms of component architecture. Binding to context properties was only possible for instance methods that inherited from a specific type, and intercession was only possible in inter-object invocations. Hence, the Fibonacci algorithm had to be split up between different objects for logging to be applied to recursive invocations. In contrast, no such concessions were required to bind aspect-based properties to component instances.

The secondary contributions of the thesis were the introduction of language-independent AOP and the use of attribute-based property selection to solving reusability problems with aspects. We take language-independence to mean allowing aspects and components to be written in a variety of languages and freely intermixed [Laf'03]. Language-independence is achieved by providing a means of expressing pointcut-advice operations that avoids language extensions. By expressing crosscutting specifications in XML, these language extensions become consistent for all programming languages used to implement aspect-based properties. The XML exploited component metadata to provide a common substrate with which to build pointcut specifications. Language-independence was illustrated in Chapter 5, when we demonstrated cross-language weaving using components and aspects written in SML.NET, C#, and VB.NET. In this evaluation every combination of aspect and component was shown to weave correctly regardless of the languages used to implement the components and aspects being woven.

Reusable aspects are those that are associated with components at deployment time rather than design time [Pic'03]. We take a novel approach in which attribute-based property selection is used to specify component bindings without the need to write or revise crosscutting specifications. Attributes-based property selection is justified in terms of obliviousness by pointing out that while attributes are annotated to component code, they do not influence the implementation of the join points. In Chapter 5, we illustrated attribute-based property selection when profiling, logging and memoization were each added to components by annotating component methods with attributes.

6.3 Future Work

Having introduced aspect-based properties, we are keen to investigate their usefulness. This involves upgrading the weaver to eliminate limits to component weaving. Next, we wish to characterise potential security problems that weaving can introduce. Also, we wish to apply aspect-based properties to implementing important enterprise application crosscutting functionalities in order to ascertain whether crosscutting mechanisms besides pointcut-advice are required for aspect-based properties to support a range of important crosscutting functionality. Finally, we wish to examine new ways of generating XML-based crosscutting specifications.

6.3.1 Upgrading the Weaver Implementation

In providing a prototype weaver that supports aspect-based properties, we were interested in implementing a prototype in as little time as possible. The choice of the CLI platform satisfied language-independence requirements in terms of multilingual support for component implementation and source code annotation. However, the novelty of the CLI platform meant that tool support was relatively weak. For instance, no byte code instrumentation packages existed for the CLI, whereas packages such as the Binary Component Engineering Library [Dah'99] has been available for sometime for the J2SE platform. Thus, while our weaver design included support for all kinds of after returning advice, to speed development time we limited support to the implementation of after returning advice. Tool support for the CLI has advanced considerably with byte code instrumentation packages now available. Using these packages for code generation should allow us to fill gaps in our weavers support for aspect-based properties.

A more pressing problem is that our prototype weaver does not handle interdependent components properly. Currently, weaving one assembly prevents the assemblies upon which it depends from being woven. This problem also relates to the lack of tool support for the CLI platform, and specifically the fact that there is not an API for modifying an existing assembly. To provide such support, our system manually recreates existing assemblies through careful examination of the original assembly with the CLIFile Reader Library [Cis'02]. However, in examining the original assembly, the CLIFile Reader loads types upon which an assembly is dependent. Loaded types cannot be manipulated by the byte code instrumentation used in our system, nor does the CLI allow types to be unloaded. We would like to eliminate this loading so that the weaver is able to ignore dependencies between components during weaving.

6.3.2 Security Implications

The security implications of binding third-party code to existing assemblies have not been investigated. The clear-box crosscutting available with aspect-based properties allows intra-method changes to components. With such tight coupling, trust becomes an issue. Currently, the weaver recreates a new assembly, but it does not verify the integrity of the original. Also, it is possible to revise code to change the basis of authorization decisions. For instance, custom crosscutting could be used to modify calls to a `System.Security.Principal.WindowsPrincipal` object that verifies user group membership. The groups for which authorization is approved could be changed, or the authorization check could be bypassed altogether.

6.3.3 Looking at a Hybrid Crosscutting Mechanism

We would like to know if other crosscutting mechanisms should be introduced to aspect-based properties. The set of aspect-based properties in this thesis did not address crosscutting functionality typical of enterprise applications such as persistence, authorization and support for transactions that have been addressed elsewhere with an aspect-oriented mechanism [Coh'04]. It would be of interest to verify that they could be supported with aspect-based properties. However, it should be pointed out that, in [Coh'04], these crosscutting functionalities were implemented with a weaving architecture that included a mixin mechanism. An aspect-based property implementation of these concerns would give us a clearer idea of whether mixins or additional mechanisms are truly required or whether they are only necessary for the underlying weaver.

6.3.4 Exploring New Ways of Expressing crosscutting Functionality

We would like to investigate new approaches to writing crosscutting functionality that exploit XML as a backend. In our evaluation, we noted that writing crosscutting specifications in XML manually was error prone, and custom crosscutting in particular did not provide a good indication of where aspect-based properties influenced component behaviour. We have recently noted alternatives to expressing crosscutting semantics. For instance, attributes have been used to embed crosscutting functionality in component code [Bla'03], and LOOM.NET [Sch'03] provided a visual editor to do the same. In our experiments we have found it relatively easy to map GUI and attribute-based pointcut-advice declarations to XML, and we would like to experiment further with such systems as there is little discussion on this topic in the literature.

Appendix A – Aspect-Based Property XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ax="http://aosd.dsg.cs.tcd.ie/XMLSchema"
  targetNamespace="http://aosd.dsg.cs.tcd.ie/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
   Test bed schema for figuring out XML for AspectJ.
   Copyright 2002-2003 Donal Lafferty,
   Distributed System Group,
   Trinity College, Dublin,
   All rights reserved.
  </xsd:documentation>
  </xsd:annotation>
  <xsd:simpleType name="identifier">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[a-zA-Z$_][a-zA-Z$_0-9]*" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="empty">
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="identifier_pattern">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[a-zA-Z$_\*][a-zA-Z$_0-9\*]*" />
    </xsd:restriction>
  </xsd:simpleType>
  <!--
<xsd:complexType name="pointcut">
  <xsd:element name="primitive"      type="ax:primitive_pointcut" minOccurs="0"
maxOccurs="unbounded"/>
  <xsd:element name="logical_op"     type="ax:binary_logical_op"  minOccurs="0"
maxOccurs="unbounded"/>
  <xsd:element name="pointcut_grp"   type="ax:pointcut"           minOccurs="0"
maxOccurs="unbounded"/>
</xsd:complexType>
-->
  <xsd:complexType name="access_modifier">
    <xsd:choice>
      <xsd:element name="public" type="ax:empty" />
      <xsd:element name="private" type="ax:empty" />
      <xsd:element name="protected" type="ax:empty" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:simpleType name="var_type">
    <xsd:restriction base="xsd:string">
      <!--     identifier is  ([a-zA-Z$_][a-zA-Z$_0-9]*)
       "." identifier is  (\.[a-zA-Z$_][a-zA-Z$_0-9]*)*(\[\]) -->
      <xsd:pattern value="([a-zA-Z$_][a-zA-Z$_0-9]*)([\.|\+][a-zA-Z$_][a-zA-Z$_0-
9]*)*(\[\])?" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="local_ref">
    <xsd:sequence>
      <xsd:element name="var_type" type="ax:var_type" />
      <xsd:element name="var_name" type="ax:identifier" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="empty_w_not">
    <xsd:complexContent>
      <xsd:extension base="ax:empty">
        <xsd:attribute name="logical_not" type="xsd:boolean" default="false" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="modifier_spec">
    <xsd:choice>
      <xsd:element name="static" type="ax:empty_w_not" />
      <xsd:element name="abstract" type="ax:empty_w_not" />
      <xsd:element name="synchronized" type="ax:empty_w_not" />
      <xsd:element name="volatile" type="ax:empty_w_not" />
      <xsd:element name="native" type="ax:empty_w_not" />
      <xsd:element name="final" type="ax:empty_w_not" />
      <xsd:element name="public" type="ax:empty_w_not" />
      <xsd:element name="protected" type="ax:empty_w_not" />
      <xsd:element name="private" type="ax:empty_w_not" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:simpleType name="type_name">
    <xsd:restriction base="xsd:string">
      <!-- regular expression for an identifier pattern is:
         ([a-zA-Z$_\*][a-zA-Z$_0-9\*]*)
         type_name is
         <identifier> [.<identifier>]* [ \[\] ]?

         .NET nested types are give as 'encloser' '+' 'nested', hence
         regular expression needs a '+' where a period can appear.
    -->
      <xsd:pattern value="([a-zA-Z$_\*][a-zA-Z$_0-9\*]*)([\.|\.\.|\+][a-zA-
Z$_\*][a-zA-Z$_0-9\*]*)*(\+)?(\[\])?" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="type_pattern">
    <xsd:choice>
      <xsd:element name="attribute" type="ax:type_name" />
      <xsd:element name="type_name" type="ax:type_name" />
      <xsd:element name="or">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="type_pattern" type="ax:type_pattern" minOccurs="2"
                         maxOccurs="2" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="and">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="type_pattern" type="ax:type_pattern" minOccurs="2"
                         maxOccurs="2" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
    <xsd:attribute name="logical_not" type="xsd:boolean" default="false" />
  </xsd:complexType>
  <xsd:complexType name="field_signature">
    <xsd:sequence>
      <xsd:element name="modifier_spec" type="ax:modifier_spec" minOccurs="0"
                   maxOccurs="unbounded" />
      <xsd:element name="field_type" type="ax:type_pattern" />
      <xsd:element name="join_point_type" type="ax:type_pattern" />
      <xsd:element name="field_name" type="ax:identifier_pattern" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="parameter_types">
    <xsd:choice>
      <xsd:element name="parameter" type="ax:type_pattern" minOccurs="0"
                   maxOccurs="unbounded" />
      <xsd:element name="param_wildcard" type="ax:empty" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="method_signature">
    <xsd:sequence>
      <xsd:element name="modifier_spec" type="ax:modifier_spec" minOccurs="0"
```

```
                 maxOccurs="unbounded" />
    <xsd:element name="return_type" type="ax:type_pattern" minOccurs="0" />
    <xsd:element name="join_point_type" type="ax:type_pattern" minOccurs="0" />
    <xsd:element name="method_name" type="ax:identifier_pattern" />
    <xsd:element name="parameters" type="ax:parameter_types" />
    <xsd:element name="throws" type="ax:type_pattern" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="advice_modifier">
  <xsd:sequence>
    <xsd:element name="strictfp" type="ax:empty" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="behaviour">
  <xsd:sequence>
    <xsd:element name="name" type="ax:identifier" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="formal_params">
  <xsd:sequence>
    <xsd:element name="local_refs" type="ax:local_ref" minOccurs="0"
                 maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="java_modifiers">
  <xsd:choice>
    <xsd:element name="static" type="ax:empty" />
    <xsd:element name="abstract" type="ax:empty" />
    <xsd:element name="synchronized" type="ax:empty" />
    <xsd:element name="volatile" type="ax:empty" />
    <xsd:element name="native" type="ax:empty" />
    <xsd:element name="final" type="ax:empty" />
    <xsd:element name="accessibility" type="ax:access_modifier" />
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="intro_modifiers">
  <xsd:choice>
    <xsd:element name="static" type="ax:empty_w_not" />
    <xsd:element name="synchronized" type="ax:empty_w_not" />
    <xsd:element name="volatile" type="ax:empty_w_not" />
    <xsd:element name="native" type="ax:empty_w_not" />
    <xsd:element name="final" type="ax:empty_w_not" />
    <xsd:element name="public" type="ax:empty_w_not" />
    <xsd:element name="private" type="ax:empty_w_not" />
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="method_introduction">
  <xsd:sequence>
    <xsd:element name="modifiers" type="ax:intro_modifiers" />
    <xsd:element name="return_type" type="ax:var_type" />
    <xsd:element name="target" type="ax:type_pattern" />
    <xsd:element name="name" type="ax:identifier" />
    <xsd:element name="formal_params" type="ax:formal_params" />
    <xsd:element name="behaviour" type="ax:behaviour" minOccurs="0" />
  </xsd:sequence>
  <xsd:attribute name="abstract" type="xsd:boolean" default="false" />
</xsd:complexType>
<xsd:complexType name="field_reference">
  <xsd:sequence>
    <xsd:element name="type" type="ax:var_type" />
    <xsd:element name="name" type="ax:identifier" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="field_introduction">
  <xsd:sequence>
    <xsd:element name="modifiers" type="ax:intro_modifiers" />
    <xsd:element name="field_type" type="ax:var_type" />
    <xsd:element name="target" type="ax:type_pattern" />
    <xsd:element name="name" type="ax:identifier" />
    <xsd:element name="field_init" type="ax:field_reference" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="super_type_introduction">
  <xsd:sequence>
    <xsd:element name="target" type="ax:type_pattern" />
    <xsd:choice>
      <xsd:element name="extends" type="ax:var_type" maxOccurs="unbounded" />
      <xsd:element name="implements" type="ax:var_type" maxOccurs="unbounded"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="signature">
  <xsd:choice>
    <xsd:element name="attribute" type="ax:type_name" />
    <xsd:element name="method_signature" type="ax:method_signature" />
    <xsd:element name="field_signature" type="ax:field_signature" />
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="type_pattern_or_id">
  <xsd:choice>
    <xsd:element name="pattern" type="ax:type_pattern" />
    <xsd:element name="formal_parameter_name" type="ax:identifier" />
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="primitive_pointcut">
  <xsd:choice>
    <xsd:element name="call" type="ax:signature" />
    <xsd:element name="execution" type="ax:signature" />
    <xsd:element name="get" type="ax:signature" />
    <xsd:element name="set" type="ax:signature" />
    <xsd:element name="initialization" type="ax:signature" />
    <xsd:element name="withincode" type="ax:signature" />
    <xsd:element name="handler" type="ax:type_pattern" />
    <xsd:element name="within" type="ax:type_pattern" />
    <xsd:element name="staticinitialization" type="ax:type_pattern" />
    <xsd:element name="cflow" type="ax:pointcut" />
    <xsd:element name="cflowbelow" type="ax:pointcut" />
    <xsd:element name="this" type="ax:type_pattern_or_id" />
    <xsd:element name="target" type="ax:type_pattern_or_id" />
    <xsd:element name="args">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="parameter" type="ax:type_pattern_or_id"
                       minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="pointcutId">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="ax:identifier" />
          <xsd:element name="parameter" type="ax:type_pattern_or_id"
                       minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="conditional_if" type="xsd:string" />
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="pointcut">
  <xsd:choice>
    <xsd:element name="primitive" type="ax:primitive_pointcut" />
    <xsd:element name="or">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="pointcut" type="ax:pointcut" minOccurs="2"
                       maxOccurs="2" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="and">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="pointcut" type="ax:pointcut" minOccurs="2"
                       maxOccurs="2" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
  <xsd:attribute name="logical_not" type="xsd:boolean" default="false" />
</xsd:complexType>
<xsd:complexType name="around_advice">
  <xsd:sequence>
    <xsd:element name="modifiers" type="ax:advice_modifier" minOccurs="0" />
    <xsd:element name="return_type" type="ax:var_type" />
    <xsd:element name="formal_param" type="ax:local_ref" minOccurs="0"
                 maxOccurs="unbounded" />
    <xsd:element name="throwing_params" type="ax:local_ref" minOccurs="0" />
    <xsd:element name="pointcut" type="ax:pointcut" />
    <xsd:element name="behaviour" type="ax:behaviour" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="after_advice">
  <xsd:sequence>
    <xsd:element name="modifiers" type="ax:advice_modifier" minOccurs="0" />
    <xsd:element name="formal_param" type="ax:local_ref" minOccurs="0"
                 maxOccurs="unbounded" />
    <xsd:element name="returning_params" type="ax:local_ref" minOccurs="0" />
    <xsd:element name="throwing_params" type="ax:local_ref" minOccurs="0" />
    <xsd:element name="pointcut" type="ax:pointcut" />
    <xsd:element name="behaviour" type="ax:behaviour" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="before_advice">
  <xsd:sequence>
    <xsd:element name="modifiers" type="ax:advice_modifier" minOccurs="0" />
    <xsd:element name="formal_param" type="ax:local_ref" minOccurs="0"
                 maxOccurs="unbounded" />
    <xsd:element name="pointcut" type="ax:pointcut" />
    <xsd:element name="behaviour" type="ax:behaviour" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="advice">
  <xsd:choice>
    <xsd:element name="before" type="ax:before_advice" />
    <xsd:element name="after" type="ax:after_advice" />
    <xsd:element name="around" type="ax:around_advice" />
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="softened_exception_introduction">
  <xsd:sequence>
    <xsd:element name="soft_exception_type" type="ax:var_type" />
    <xsd:element name="pointcut" type="ax:pointcut" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="error_warning_introduction">
  <xsd:sequence>
    <xsd:element name="target" type="ax:pointcut" />
    <xsd:choice>
      <xsd:element name="error" type="xsd:string" />
      <xsd:element name="warning" type="xsd:string" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="member_introduction">
  <xsd:choice>
    <xsd:element name="method" type="ax:method_introduction" />
    <xsd:element name="field" type="ax:field_introduction" />
  </xsd:choice>
</xsd:complexType>
```

```
  <xsd:complexType name="introduction">
    <xsd:choice>
      <xsd:element name="member" type="ax:member_introduction" />
      <xsd:element name="super_type" type="ax:super_type_introduction" />
      <xsd:element name="error_warning" type="ax:error_warning_introduction" />
      <xsd:element name="softened_exception"
                   type="ax:softened_exception_introduction" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="instantiation_info">
    <xsd:choice>
      <xsd:element name="issingleton" type="ax:empty" />
      <xsd:element name="perthis" type="ax:pointcut" />
      <xsd:element name="pertarget" type="ax:pointcut" />
      <xsd:element name="percflow" type="ax:pointcut" />
      <xsd:element name="percflowbelow" type="ax:pointcut" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:element name="aspect">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="ax:identifier" />
        <xsd:element name="assembly" type="ax:identifier" />
        <xsd:element name="type" type="ax:var_type" />
        <xsd:element name="instantiation" type="ax:instantiation_info"
                     minOccurs="0" />
        <xsd:element name="domination" type="ax:type_pattern" minOccurs="0" />
        <xsd:element name="body" type="ax:aspect_body" />
      </xsd:sequence>
      <xsd:attribute name="privileged" type="xsd:boolean" default="true" />
      <xsd:attribute name="abstract" type="xsd:boolean" default="true" />
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="named_pointcut">
    <xsd:sequence>
      <xsd:element name="modifier" type="ax:access_modifier" />
      <xsd:element name="name" type="ax:identifier" />
      <xsd:element name="local_var_ref" type="ax:local_ref" minOccurs="0"
                   maxOccurs="unbounded" />
      <!-- Pointcut specification not required in the case that a
         this is an abstract named pointcut. -->
      <xsd:element name="pointcut" type="ax:pointcut" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="abstract" type="xsd:boolean" default="false" />
    <xsd:attribute name="final" type="xsd:boolean" default="false" />
  </xsd:complexType>
  <xsd:complexType name="aspect_body">
    <xsd:sequence>
      <xsd:element name="item" type="ax:crosscutting_statement" minOccurs="0"
                   maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="crosscutting_statement">
    <xsd:choice>
      <xsd:element name="introduction" type="ax:introduction" />
      <xsd:element name="advice" type="ax:advice" />
      <xsd:element name="named_pointcut" type="ax:named_pointcut" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="Test">
    <xsd:choice>
      <xsd:element name="Recurse" type="ax:Test" />
      <xsd:element name="Foo" type="xsd:string" />
      <xsd:element name="Bar" type="xsd:string" />
    </xsd:choice>
  </xsd:complexType>
</xsd:schema>
```

Bibliography

[Aks'92] Aksit, M., Bergmans, L. and Vural, S., An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In *European Conference on Object-Oriented Programming (ECOOP'92)*, (1992).

[Asp'00] AspectJ.org. *AspectJ Home Page*, website, http://www.aspectj.org, 2000.

[Ber'99] Bergmans, L. Aspects of Middleware-based Telematics Services, Centre for Telematics and Information Technology, University of Twente, 1999, pp.48.

[Ber'94] Bergmans, L. *The Composition-Filters Object Model*. PhD Thesis, Department of Computer Science, University of Twente, 1994.

[Ber'01] Bergmans, L. and Aksit, M. "Composing Crosscutting Concerns Using Composition Filters". *Communications of the ACM*, *44* (10), October 2001, pp.51-57.

[Ber'04a] Bergmans, L. and Aksit, M. Principles of Design Rationale of Composition Filters. in Aksit, M., Clarke, S., Elrad, T. and Filman, R. eds. *Aspect-Oriented Software Development*, Addison-Wesley, 2004a.

[Ber'04b] Bergmans, L., Nagy, I., Gulesir, G. and Aksit, M. Compose*: Language-independent Aspects in .NET, Demonstration, 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), 2004b.

[Bla'03] Blackstock, M. Aspect Weaving with C# and .NET, University of British Columbia, Vancouver, 2003.

[Blo'03] Bloch, J. *JSR-000175 A Metadata Facility for the JavaTM Programming Language*, http://jcp.org/aboutJava/communityprocess/review/jsr175/, 2003.

[Bon'04a] Bonér, J. and Vasseur, A., AspectWerkz for Dynamic Aspect-Oriented Programming. In *Tutorial #2* in *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, (Lancaster, UK, 2004a).

[Bon'04b] Bonér, J. and Vasseur, A. *AspectWerkz: Simple, dynamic, lightweight and powerful AOP for Java*, http://aspwectwerkz.codehaus.org/, 2004b.

[Boo'94] Booch, G. *Object-oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, California, 1994.

[Bry'02] Bryant, A. and Feldt, R. *AspectR - Simple aspect-oriented programming in Ruby*, http://aspectr.sourceforge.net/, 2002.

[Cah'02] Cahill, V. and Lafferty, D. *Learning to Program the Object-Oriented Way with C#*. Springer-Verlag UK, London, 2002.

[Car'97] Cardelli, L., Program Fragments, Linking, and Modularization. In *24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'97)*, (Paris, France, 1997), ACM, pp.266 - 277.

[Car'01] Caro, P.S. *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*. M.Sc. Thesis, Computer Science, Vrije Universiteit Brussel, Brussels, 2001.

[Chi'00] Chiba, S., Load-time Structural Reflection in Java. In *European Conference on Object-Oriented Programming (ECOOP 2000)*, (Cannes, France, 2000), Springer Verlag, pp.313-336.

[Cis'02] Cisternino, A. *CLIFileReader Library*, C# Source Code, http://dotnet.di.unipi.it/MultipleContentView.aspx?code=103, 2002.

[Cle'04] Clement, A., Colyer, A., Kersten, M., Park, J.W. and Chapman, M. *AspectJ Development Tools (AJDT) subproject*, http://www.eclipse.org/ajdt/, 2004.

[Cli'00] Clifton, C., Leavens, G., Chambers, C. and Millstein, T., MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, (Minneapolis, MN, USA, 2000), pp.130-145.

[Coa'01] Coady, Y., Smolyn, G. and Kiczales, G. *AspectC*, http://www.cs.ubc.ca/labs/spl/projects/aspectc.html, 2001.

[Coh'98] Cohen, G., Chase, J. and Kaminsky, D., Automatic Program Transformation with JOIE. In *USENIX Annual Technical Conference '98,* (1998).

[Coh'04] Cohen, T. and Gil, J., AspectJ2EE = AOP + J2EE. In *European Conference on Object-Oriented Programming (ECOOP 2004)*, (Oslo, Norway, 2004), Springer, pp.219-243.

[Cos'03] Costanza, P. "Dynamically scoped functions as the essence of AOP". *ACM SIGPLAN Notices*, *38* (8), August 2003, pp.29 - 36.

[Cza'00] Czarnecki, K. and Eisenecker, U.W. *Generative Programming*. Addison-Wesley, New York, 2000.

[Dah'99] Dahm, Byte Code Engineering. In *Java-Informations-Tage*, (Düsseldorf, 1999), pp.267-277.

[DeM'03] DeMichiel, L.G. *Enterprise JavaBeans Specification, Version 2.1*, Sun Microsystems, 2003.

[Dij'76] Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, London, 1976.

[Duc'02] Duclos, F., Estublier, J. and Morat, P., Describing and Using Non Functional Aspects in Component Based Applications. In *1st International Conference on Aspect-Oriented Software Development*, (Enschede, The Netherlands, 2002), pp.65 - 75.

[Ecm'03a] ECMA International. *Standard ECMA-334 C# Language Specification*, ECMA Standard, http://www.ecma-international.org/publications/standards/ecma-334.htm, 2003a.

[Ecm'03b] ECMA International. *Standard ECMA-335 Common Language Infrastructure (CLI)*, ECMA Standard, http://www.ecma-international.org/publications/standards/ecma-335.htm, 2003b.

[Edd'99] Eddon, G. and Eddon, H. *Inside COM+*. Microsoft Press, 1999.

[Elr'01] Elrad, T., Filman, R.E. and Bader, A. "Aspect-oriented Programming". *Communications of the ACM, 44* (10), October 2001, pp.29-32.

[Est'02] Estier, T. *What is BNF notation?*, Website, http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html, 2002.

[Fal'01] Fallside, D.C. *XML Schema Part 0: Primer*, W3C Recommendation, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, 2001.

[Fil'00] Filman, R.E. and Friedman, D.P., Aspect-Oriented Programming is Quantification and Obliviousness. In *Advanced Separation of Concerns Workshop* in *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, (Minneapolis, USA, 2000).

[Fin'98] Findler, R. and Flatt, M., Modular Object-oriented Programming with Units and Mixins. In *International Conference on Functional Programming (ICFP'98)*, (Baltimore, United States, 1998), ACM, pp.94 - 104.

[Fri'01] Friedman, D., Wand, M. and Haynes, C.T. *Essentials of Programming Languages*. MIT Press, London, England, 2001.

[Gam'94] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*. Addison-Wesley, Don Mills, Ontario, 1994.

[Gar'03] Garcia, C.F.N. *Compose * A Runtime for the .Net Platform*. MSc Thesis, Computer Science, Vrije Universiteit Brussel, Brussels, 2003.

[Gos'00] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification*. Addison-Wesley, London, 2000.

[Gra'04] Gray, J. and Roychoudhury, S., A Technique for Constructing Aspect Weavers Using a Program Transformation Engine. In *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, (Lancaster, UK, 2004), ACM.

[Gra'97] Gray, S.D. and Lievano, R.A. *Microsoft Transaction Server 2.0*, 1997.

[Han'01] Hannemann, J. and Kiczales, G., Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns* in *International Conference on Software Engineering (ICSE) 2001*, (Toronto, 2001).

[Har'93] Harrison, W. and Ossher, H., Subject-Oriented Programming (A Critique of Pure Objects). In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, (Washington, DC, 1993), ACM Press, pp.411-428.

[Hil'04] Hilsdale, E. and Hugunin, J., Advice Weaving in AspectJ. In *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, (Lancaster, UK, 2004), ACM.

[Hor'00] Hors, A.L., Hégaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M. and Byrne, S. *Document Object Model (DOM) Level 2 Core Specification*, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113, 2000.

[IBM'00a]IBM. *Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces*, website, http://www.research.ibm.com/hyperspace/, 2000a.

[IBM'00b]IBM. *Subject-oriented programming,* website, http://www.research.ibm.com/sop/, 2000b.

[jBo'01] jBoss. *jBoss Container*, http://www.jboss.org, 2001.

[Mat'96] Kaplan, M., Ossher, H., Harrison, W. and Kruskal, V., Subject-oriented design and the Watson Subject Compiler. In *Workshop on Subjects and Viewpoints Throughout the Lifecycle* in *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, (Atlanta, Georgia, 1996).

[Kel'98] Keller, R. and Hölzle, U., Binary Component Adaptation. In *European Conference on Object-Oriented Programming (ECOOP'98)*, (Brussels, Belgium, 1998), Springer-Verlag, pp.309-329.

[Ken'03] Kennedy, A., Russo, C. and Benton, N. *SML.NET 1.1 User Guide*, Online Book, http://wwww.cl.cam.ac.uk/Research/TSG/SMLNET/smlnet.pdf, 2003.

[Kic'01a] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. "Getting Started with AspectJ". *Communications of the ACM, 44* (10), October 2001a, pp.59-65.

[Kic'01b] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP 2001)*, (Budapest, Hungary, 2001b), Springer-Verlag, pp.327-355.

[Kic'97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M. and Irwin, J., Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, (Jyväskylä, Finland, 1997), Springer-Verlag, pp.220-242.

[Kim'02] Kim, H. and Clarke, S., The relevance of AOP to an applications programmer in an EJB environment. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)* in *1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, (Enschede, The Netherlands, 2002).

[Kni'01a] Kniesel, G., Costanza, P. and Austermann, M., Independent Extensibility for Aspect-Oriented Systems. In *Advanced Separation of Concerns Workshop* in *European Conference on Object-Oriented Programming (ECOOP 2001)*, (Budapest, Hungary, 2001a).

[Kni'01b] Kniesel, G., Costanza, P. and Austermann, M., JMangler - A Framework for Load-Time Transformation of Java Class Files. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, (Florence, Italy, 2001b).

[Lad'03] Laddad, R. *AspectJ In Action*. Manning, Greenwich, CT, 2003.

[Laf'02a] Lafferty, D. *W3C XML Schema for AspectJ Aspects*, XML Schema, http://aosd.dsg.cs.tcd.ie/XMLSchema/aspect_Schema.xsd, 2002a.

[Laf'02b] Lafferty, D. *W3C XML Schema for AspectJ aspects survey of AspectJ grammar*, http://www.dsg.cs.tcd.ie/index.php?category_id=144, 2002b.

[Laf'03] Lafferty, D. and Cahill, V., Language-Independent Aspect-Oriented Programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, (Anaheim, California, USA, 2003).

[Lam'02] Lam, J. Cross Language Aspect Weaving, Demonstration, AOSD 2002, Enschede, 2002.

[Lid'02] Lidin, S. *Inside Microsoft .NET IL Assembler*. Microsoft Press, Redmond, Washington, 2002.

[Lie'00] Lieberherr, K. *Demeter Home Page*, website, http://www.ccs.neu.edu/home/lieber/demeter.html, 2000.

[Lie'99] Lieberherr, K., Lorenz, D. and Mezini, M. Programming with Aspectual Components, College of Computer Science, Northeastern University, Boston, MA, 1999.

[Lie'01] Lieberherr, K., Orleans, D. and Ovlinger, J. "Aspect-Oriented Programming with Adaptive Methods". *Communications of the ACM*, *44* (10), October 2001, pp.39-41.

[Lie'04] Lieberherr, K., Patt-Shamir, B. and Orleans, D. "Traversals of Object Structures: Specification and Efficient Implementation". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *26* (2), March 2004, pp.370–412.

[Lie'96] Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[Lip'99] Lippert, M. and Lopes, C.V. A Study on Exception Detection and Handling Using Aspect-Oriented Programming, Xerox PARC, 1999.

[Lop'97] Lopes, C.V. and Kiczales, G. D: A Language Framework for Distributed Programming, Xerox PARC, 1997.

[Mas'03] Masuhara, H. and Kiczales, G., Modeling Crosscutting in Aspect-Oriented Mechanisms. In *European Conference on Object-Oriented Programming (ECOOP 2003)*, (Darmstadt, Germany, 2003), Springer-Verlag.

[Mat'93] Matsuoka, S. and A. Yonezawa. Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. in Agha, G., Wegner, P. and Yonezawa, A. eds. *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993, pp.107-150.

[Men'97] Mendhekar, A., Kiczales, G. and Lamping, J. RG: A Case-Study for Aspect-Oriented Programming, Xerox PARC, 1997.

[Mic'04a] Microsoft. *ILDASM (Microsoft .NET Framework IL Disassembler)*, Development Tool, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfpeverifytoolpeverifyexe.asp, 2004a.

[Mic'04b] Microsoft. *Microsoft .NET Framework SDK*, Website, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/NETFxAnchor.asp, 2004b.

[Mic'04c] Microsoft. *PEVerify Tool (Peverify.exe)*, Development Tool, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfpeverifytoolpeverifyexe.asp, 2004c.

[Mic'01] Microsoft. Standard ECMA-334 C# Language Specification, ECMA International - European association for standardizing information and communication systems, 2001.

[Mic'04d] Microsoft. *Visual Basic Development Center*, MSDN website, http://msdn.microsoft.com/vbasic, 2004d.

[Mic'04e] Microsoft. *Visual J# .NET Developer Center*, MSDN website, http://msdn.microsoft.com/vjsharp/, 2004e.

[Mil'99] Millstein, T. and Chambers, C., Modular Statically Typed Multimethods. In *European Conference on Object-Oriented Programming (ECOOP'99)*, (Lisbon, Portugal, 1999), Springer, pp.279-303.

[Moo'86] Moon, D., Object-oriented programming with Flavors. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, (1986), pp.1-8.

[Mur'99] Murphy, G.C., Walker, R.J. and Baniassad, E.L.A. "Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming". *IEEE Transactions on Software Engineering, 25* (4) 1999, pp.435-455.

[New'02] Newkirk, J. and Vorontsov, A.A. "How .NET's Custom Attributes Affect Design". *IEEE Software*, *19* (5), September/October 2002, pp.18-20.

[OMG] OMG. *Corba Component Model*, http://www.omg.org.

[OMG'03] OMG. *OMG Unified Modeling Language Specification, Version 1.5, formal/03-03-01*, website, http://www.omg.org/technology/documents/formal/uml.htm, 2003.

[Orl'04] Orleans, D. and Lieberherr, K. *DemeterJ*, website, http://www.ccs.neu.edu.research/demeter/DemeterJava, 2004.

[Orl'01] Orleans, D. and Lieberherr, K., DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, (Kyoto, Japan, 2001), Springer Verlag, pp.73-80.

[Oss'94] Ossher, H., Harrison, W., Budinsky, F. and Simmonds, I., Subject-Oriented Programming: Supporting Decentralized Development of Objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, (1994).

[Oss'95] Ossher, H., Kaplan, M., Harrison, W., Katz, A. and Kruskal, V., Subject-Oriented Composition Rules. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95),* (Austin, Texas, USA, 1995), ACM Press, pp.235-250.

[Oss'01a] Ossher, H. and Tarr, P., Hyper/J: Multi-dimensional separation of concerns for Java. In *23rd International Conference on Software Engineering (ICSE 2001),* (2001a), pp.729-730.

[Oss'01b] Ossher, H. and Tarr, P. Multi-dimensional separation of concerns and the hyperspace approach. in Aksit, M. ed. *Software Architectures and Component Technology*, Kluwer, 2001b.

[Oss'01c] Ossher, H. and Tarr, P. "Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software". *Communications of the ACM*, *44* (10) 2001c, pp.43-50.

[Pic'03] Pichler, R., Ostermann, K. and Mezini, M. "On Aspectualizing Component Models". *Software Practice and Experience*, *33* (10), August 2003, pp.957-974.

[Ras'01] Rashid, A., Hybrid Approach to Separation of Concerns: The Story of SADES. In *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION'01),* (2001), Springer-Verlag, pp.231 - 249.

[Ras'03] Rashid, A. and Chitchyan, R. Persistence as an Aspect *2nd international conference on Aspect-Oriented Software Development (AOSD 2003),* ACM, Boston, Massachusetts, 2003, pp.120 - 129.

[Rom'02] Roman, E., Ambler, S. and Jewell, T. *Mastering Enterprise JavaBeans*. John Wiley & Sons, Inc., Toronto, 2002.

[Sab'04] Sabbah, D.D. Keynote: [CHECK], Demonstration, 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), 2004.

[Sch'03] Schult, W. *LOOM.NET*, http://www.dcl.hpi.uni-potsdam.de/cms/research/loom/, 2003.

[Sch'02] Schult, W. and Polze, A., Aspect-Oriented Programming with C# and .NET. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, (Washington, DC, 2002), IEEE Computer Society Press, pp.241-248.

[Shu'02] Shukla, D., Fell, S. and Sells, C. "AOP: Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse". *MSDN Magazine*, *17* (3) 2002, pp.

[Soa'02] Soares, S., Laureano, E. and Borba, P., Implementing Distribution and Persistence Aspects with AspectJ. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, (Seattle, Washington, USA, 2002), ACM Press, pp.174 - 190.

[Spi'04] Spinczyk, O., Urban, M., Gal, A. and Lohmann, D. *The Home of AspectC++*, website, http://www.aspectc.org/, 2004.

[Sun'04] Sun Microsystems Inc. *Javadoc*, http://java.sun.com/j2se/javadoc, 2004.

[Sun'00] Sun Microsystems Inc. *JDC Tech Tips: June 27, 200*, Website, http://java.sun.com/developer/TechTips/2000/tt0627.html, 2000.

[Szy'02] Szyperski, C., Gruntz, D. and Murer, S. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, London, 2002.

[Tar'99] Tarr, P., Ossher, H., Harrison, W. and Stanley M. Sutton, J., N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *21st International Conference on Software Engineering (ICSE'99)*, (Los Angeles, USA, 1999), IEEE Computer Society Press, pp.107-119.

[Asp'02] The AspectJ Team. *The AspectJ Programming Guide (V1.0.6)*, http://download.eclipse.org/technology/ajdt/aspectj-docs-1.0.6.tgz, 2002.

[Tou'03] Tourwé, T., Brichau, J. and Gybels, K., On the Existence of the AOSD Evolution Paradox. In *Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT)* in *2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, (Boston, USA, 2003).

[TRE'04] TRESE. *Aspect-Oriented Research on Composition Filters homepage*, http://trese.cs.utwente.nl/composition_filters/, 2004.

[Ung'87] Ungar, D. and Smith, R.B., Self: The Power of Simplicity. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, (Orlando, Florida, USA, 1987), ACM Press, pp.227-242.

[Vli'01] van der Vlist, E. *Comparing XML Schema Languages*, Website, http://www.xml.com/lpt/a/2001/12/12/schemacompare.html, 2001.

[van'04] van Heesch, D. *Doxygen*, http://www.doxygen.org, 2004.

[XML'01] XML.COM. *XML.com*, http://www.xml.com, 2001.