Aspects of Incremental Programming

by

Brian S. de Alwis

B.Math., University of Waterloo, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

April 2002

© Brian S. de Alwis, 2002
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date 2002/04/18
Abstract

This thesis presents the design and implementation of Apostle, an extension to the Smalltalk programming language providing Aspect-Oriented Programming (AOP) functionality. Smalltalk is more than simply a language, it also encompasses a dynamic and incremental environment. Any extension to Smalltalk must preserve the properties expected of these environments, such as their immediate, incremental, and seamless natures.

As such, the design process of Apostle entails more than proposing new language constructs and their implementation. We examine three parts of the system:

- the design of the language extensions;
- the implementation;
- applying the language extensions to some real-world problems to evaluate their usefulness.
# Contents

Abstract ii

Contents iii

List of Tables vii

List of Figures viii

## 1 Introduction

1.1 Thesis Statement ........................................... 1
1.2 Motivation: Capturing Crosscutting Concerns in Smalltalk .......... 1
  1.2.1 Alternative Structuring Solutions .......................... 2
1.3 Proposed Solution: Aspect-Oriented Programming .................. 3
1.4 Goals ..................................................... 4
1.5 Research Contributions ....................................... 4
1.6 Typographic Conventions ..................................... 4

## 2 Background

2.1 Overview of Smalltalk and 'The Smalltalk Way' ................... 6
  2.1.1 The Language ............................................ 6
  2.1.2 Program Structuring Support ............................. 7
  2.1.3 The Programming Environment ............................ 7
  2.1.4 Incremental Programming .................................. 8
  2.1.5 Reflective ............................................... 8
  2.1.6 Evaluation Criteria for AOP Solutions for Smalltalk ........ 8
2.2 Prior Work Leading to AOP ................................... 9
  2.2.1 Open Classes ............................................. 9
  2.2.2 Generic Functions in the Common Lisp Object System ........ 10
  2.2.3 Programming the Meta Level .............................. 10
2.3 Aspect-Oriented Programming ................................ 10
2.3.1 AOP's Place in the World ........................................ 11
2.4 The AspectJ Model of Aspect-Oriented Programming .......... 11
  2.4.1 Interface .................................................... 12
  2.4.2 Implementation: Static Weaving ............................ 12
2.5 Previous AOP-like Solutions for Smalltalk ...................... 13
  2.5.1 Improved Structuring of Smalltalk Applications with ENVY/Manager ........................................ 13
  2.5.2 MethodWrappers ............................................. 15
2.6 Prior AOP Solutions for Smalltalk ................................ 16
  2.6.1 Andrew ...................................................... 16
  2.6.2 AOP/ST ..................................................... 16
  2.6.3 Pryor and Bastán's Metaobject Architecture for AOP ...... 17
  2.6.4 AspectS for Squeak .......................................... 17
2.7 Summary .......................................................... 18

3 Modelling Aspect-Oriented Programming In And For Smalltalk 19
  3.1 Design Constraints .............................................. 19
  3.2 Language Fundamentals ......................................... 20
    3.2.1 Join Point Model ......................................... 20
    3.2.2 A Simple Example: Expressing the Example from the Introduction ...................................... 21
    3.2.3 Naming Pointcuts .......................................... 21
    3.2.4 Storage of State Related to an Aspect ................. 22
    3.2.5 Other Aspect and Advice Types .......................... 22
    3.2.6 The Execution Context of Advice ....................... 24
    3.2.7 Affecting The Execution Context of a Join Point .... 24
    3.2.8 Summary .................................................... 24
  3.3 Discussion and Justification of Language Design .............. 25
    3.3.1 The Necessity for a Declarative Language ............. 25
    3.3.2 Justification for Expressing Pointcuts as Predicates . 26
    3.3.3 Exploring a More Smalltalk-like Pointcut Language ... 26
  3.4 Summary ........................................................ 27

4 Implementation of Apostle .......................................... 28
  4.1 Overview ...................................................... 28
  4.2 The Apostle Target Model ..................................... 30
    4.2.1 Join Points and Join Point Shadows .................... 30
    4.2.2 Advice Applicability and Dispatch at Join Point Shadows ................................................. 32
    4.2.3 Transforming Join Point Shadows: Dispatching Advice ..................................................... 33
    4.2.4 The Transformation of Advice ............................ 35
  4.3 Incremental Weaving .......................................... 36
4.3.1 Dependencies Amongst Language Elements .......................... 37
4.3.2 Types of Incremental Changes ........................................ 37
4.3.3 Support for Incremental Weaving .................................... 38
4.3.4 The Re-Weaving Algorithm .......................................... 39
4.3.5 Analysis of the Re-Weaving Algorithm ............................. 41
4.4 Summary ........................................................................... 43

5 Results and Evaluation .......................................................... 45
5.1 Three Case Studies: Using Apostle For Application Restructuring .... 45
5.1.1 Aspectizing a Model/View/Controller Application .................. 45
5.1.2 Progress Monitoring ..................................................... 49
5.1.3 Adding Unanticipated Functionality to a Closed Framework ...... 51
5.2 Subjective Evaluation ....................................................... 53

6 Conclusions and Future Work ................................................ 55
6.1 Future Work .................................................................... 55
6.1.1 Lazy Installation ......................................................... 56
6.1.2 Other Incremental Environments ..................................... 56
6.1.3 Side-Effects Arising From Incremental Changes ................. 56
6.1.4 Use and Misuse of Types in Identifying Join Points .............. 57
6.1.5 Naming Advice ............................................................ 57

Bibliography ............................................................................ 59

Appendix A A Simple Smalltalk Tutorial .................................... 63
A.1 Fundamentals .................................................................. 63
A.2 Variables and Their Assignment ........................................ 63
A.3 Sending Messages and Return Values ................................ 64
A.3.1 A Simple Example ...................................................... 64
A.3.2 A More Complex Method ............................................. 64
A.4 Blocks ............................................................................ 66
A.4.1 Out-of-Context Returns .............................................. 67
A.4.2 Special Block Messages .............................................. 67
A.5 Classes .......................................................................... 68
A.5.1 Creating Instances of a Class ....................................... 68
A.5.2 Reflection and the Metaclass Jungle .............................. 68
A.6 Other Miscellany ............................................................. 69
A.7 Summary ....................................................................... 69
# Appendix B  Specification of the Apostle Language

B.1 Declarative Specification .................................................. 70
   B.1.1 Pointcuts ................................................................. 70
   B.1.2 Naming Pointcuts and Referencing Named Pointcuts .......... 71
   B.1.3 Advice ................................................................. 71
   B.1.4 Aspects ................................................................. 72
   B.1.5 Capturing the Execution Context at a Join Point .............. 73

B.2 Apostle’s MOP: A Programmatic Interface ............................ 74
   B.2.1 Aspects ................................................................. 74
   B.2.2 Pointcuts ............................................................... 75
   B.2.3 Advice ................................................................. 76
   B.2.4 Batching Changes .................................................... 76
   B.2.5 Reflective Capabilities .............................................. 77
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Properties associated with method reception and execution join points, with indications of those determinable statically at a shadows.</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>The information tested (T) by Apostle's primitive pointcuts and possibly statically known (s). Some parts of this information is able to be exposed (E) to the advice.</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Usability Timings</td>
<td>53</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Crosscutting concern .................................................. 2
2.1 A browser showing the many extensions of the class ApAdvice by multiple applications (second of the upper panes). The third upper pane shows the method categories for this class, and the fourth the methods belonging to the selected categories. .................................................. 14
4.1 The implementation structure of Apostle. .......................... 29
4.2 The steps in an incremental weaver. ............................... 29
5.1 The ellipsis acts as a visual cue that other configurations of this button have information. .................................................. 52
Chapter 1

Introduction

1.1 Thesis Statement

This dissertation shows that Smalltalk can be extended with aspect-oriented programming (AOP) constructs, while respecting and preserving the properties expected of Smalltalk environments. While there have been prior proposals for retrofitting AOP extensions to Smalltalk, they all suffer in some regard: they are not general-purpose, these provide little information on their implementation, or — most importantly — they do not fit in with the character of the environment.

1.2 Motivation: Capturing Crosscutting Concerns in Smalltalk

One goal in building any computer system is to cleanly capture the design of the system in a modular way. Programming languages support this by providing various constructs to express and separate different software behaviours, or concerns, while also hiding unnecessary or mechanical detail. Successfully structured concerns are localized and relatively easy to reason about and understand. But some concerns are difficult to modularize cleanly with current constructs: their influence is spread across the system, and are said to crosscut the system. Left in non-modular forms, they lead to interdependencies between modules, increasing the complexity of the code and requiring yet more context to understand. Crosscutting concerns thus inhibit reuse and increase the difficulty in determining the effects or dependencies arising from changing a line of code [23].

Examples of crosscutting concerns abound. Logging is one example of a crosscutting concern, as logging calls are usually scattered and tangled throughout system code [21]. Software layers are often violated to meet performance requirements [11], or hacked to pass data through layers. Methods or variables are sometimes introduced to an object to make the object more polymorphically acceptable in another context, though they may be unre-
lated to the object’s original purpose. All are indications of crosscutting. Scattered code is found in places where it does not seem to belong, but must for lack of a better location. Collecting this code into a common location seems to be a worthy improvement.

Smalltalk, an object-oriented programming language, is not exempt from these structuring problems. This is illustrated by the following extract, taken from the simulation application discussed in Section 5.1.1.\(^1\) The progress of the simulation is reflected by a GUI, such as when a node is scheduled for execution, or actually executes. The actual simulation logic is separated from the windowing code, such that the simulator can be run stand-alone. But the window must be somehow notified of changes in the simulator. A naive implementation might cause a direct call to the ‘window’; observe the last line of the following:

```smalltalk
PercolationSimulator>>#schedule: node
| ct delta |
(self nodeEnabled: node) ifFalse: ['^self'].
node isSlow: self isNextOperationSlow.
ct := self currentTime.
delta := acceleration * (self executionTimeFor: node).
self addEvent: (SimulationEvent at: ct + delta node: node).
node scheduled. "alert the node that it has been scheduled"
window notNil ifTrue: [window drawNode: node]
```

Line 11 bears little relation to the rest of the code. This tightly couples the simulator to this window representation: the graphing code is crosscutting the simulation code.

### 1.2.1 Alternative Structuring Solutions

There are non-language constructs that attempt to render these crosscuts in a more palatable and pleasing manner, such as wrappers, subject/observers, and visitors, amongst others. Indeed, certain design patterns could be seen as relatively pleasing structuring methods for

\(^1\)Appendix A presents a simple tutorial of Smalltalk.
crosscutting concerns. But these often only mask the crosscutting by slightly decoupling the different concerns: the non-modular crosscutting is still present.

The Observer pattern [15] is often used to remove such couplings. Line 11 can be replaced with:

```
"Alert any interested parties that an event has been scheduled."
observers do: [:observer | observer eventScheduled : event]
```

While an improvement, this replacement still bears little relation to the remainder of the method: it could be safely removed without affecting the simulation. While Observer does provide some separation, the simulator code is still scattered with observer hooks.

Refactoring is often touted as the ‘correct’ solution to such structuring problems. But refactoring is often impractical due to lack of resources or time, or impossible when lacking access to or ownership over the source. It may not even be possible to find any adequate refactoring. Consider refactoring the previous simulation example: the code to notify the observers must appear somewhere. It is the presence of the observer-notification code that makes it crosscutting, not the notification implementation.

This type of problem will recur even after extensive refactoring as these structuring problems are inherent. It seems little can be done but to incorporate distasteful and seemingly-extraneous bits throughout the system.

### 1.3 Proposed Solution: Aspect-Oriented Programming

These solutions cannot modularize the crosscutting as they only treat the symptoms of the problem. These problems occur because programming languages have no facility for expressing even the simple concept ‘having finished scheduling a node, update the simulation window.’ Aspect-Oriented Programming (AOP) [30] proposes adding new constructs to do exactly that, thus capturing these crosscutting concerns in a principled manner. This is done by providing the ability to cause execution of other code at particular moments of interest in the execution of the program, without requiring ‘fingerprints’ in the original code. AOP is discussed in more detail in Section 2.3.

This thesis discusses the design and implementation of Apostle, an extension to the Smalltalk language providing AOP functionality. The Apostle language is closely modelled on that of AspectJ [28, 29], an aspect-oriented extensions to Java. But Smalltalk is very different from Java: it is a dynamically-typed language, and is tightly integrated with its development environment. Thus defining such a language involves more than specifying new constructs, as the language implementation must seem to fit the character of the environment. While there have been prior proposals for retrofitting AOP extensions to Smalltalk, they are either not general-purpose, provide little information or follow-up on their implementation, or—most importantly—do not fit in with the Smalltalk environment.
Apostle aims to succeed in all respects.

1.4 Goals

This dissertation presents:

1. a discussion of the properties of Smalltalk: the expectations held and assumed by its programmers;

2. the design of a general-purpose aspect language for Smalltalk, and a suggestion on presenting the aspect language within the dynamic Smalltalk programming environments;

3. a description of such an implementation, undertaken as a proof-of-concept to determine the necessary algorithms and constraints required to efficiently support the design;

4. an evaluation of having applied the resulting system to some real bodies of code.

1.5 Research Contributions

This thesis makes the following contributions:

• describes an implementation that fits the character of the Smalltalk environment;

• shows that an AspectJ-style language works for Smalltalk and, by extension, dynamically-typed class-based languages.

1.6 Typographic Conventions

Since Smalltalk code was intended to be viewed within a browsing environment, it has no reader-oriented declarative form. This thesis uses the following form. Individual methods, advice, and pointcuts, are presented in the following form:

```
ApPointcut>>#applicableProgramElementsFrom: loom do: block

loom applicableProgramElementsIdentifiedBy: self do: block
```

The method name follows the standard form of `Class>>#selector`. When clear from the surrounding context, the class may be omitted. Class methods are denoted by appending ‘class’ to the class name, as in ‘ApPointcut class’.

Multiple methods, advice, or pointcuts, on a class or aspect, are differentiated through use of indentation. The following example describes three methods:
Object>>#isAdvice

^false

ApPointcut>>#isAdvice

^true

ApPointcut>>#name

"Answer the name of this piece of advice; this is unique within
the definer's advice. Answer nil if this advice has no name;
it is anonymous."

^name

Again, when clear from the surrounding context, class names may be dropped.
Chapter 2

Background

Building an aspect language for Smalltalk involves more than simply porting AspectJ: there are important stylistic and cultural differences which must be accounted for. This chapter investigates some of the background required for designing and implementing such a language. Section 2.1 discusses the key properties of Smalltalk that make it successful, summarized in Section 2.1.6 as a set of requirements of any AOP solutions for Smalltalk. Section 2.2 examines some of the early research inspiring AOP. The necessary features of an aspect language are outlined in Section 2.3. AspectJ, the best-known aspect language extension and the model for Apostle, is discussed in Section 2.4. Finally sections 2.5 and 2.6 analyse the existing AOP-like or AOP solutions for Smalltalk, both from their adherence to the Smalltalk properties (Section 2.1.6), and their implementation strategies.

2.1 Overview of Smalltalk and ‘The Smalltalk Way’

Smalltalk-80 [17], or just ‘Smalltalk,’ emerged from Xerox PARC in the early 1980s [24]. It is widely credited as the first truly object-oriented language; its creator was certainly the first to coin the term “object-oriented”. Smalltalk has since been used in a diverse set of applications, from embedded devices such as the Tektronix oscilloscope [13] to major financial payroll systems such as the Chrysler Payroll System [47]. Although there are several dialects of Smalltalk, they are all variants of Smalltalk-80. This section discusses the key properties of Smalltalk that make it successful. Appendix A presents a brief tutorial on the Smalltalk language and terminology.

2.1.1 The Language

Smalltalk is a liberal language that forces very little policy, relying instead on programming convention and idiom (i.e. [2, 43]). Other languages impose such things as static
typing systems or exception systems; these are optional in Smalltalk. Even its control-flow constructs and exception system are built in the language—and so can be replaced.

Everything in Smalltalk is a first-class object, including numbers, classes and methods. It is a dynamically-typed language, meaning that types are resolved at run-time rather than at compile-time. Programming relies on published protocols, descriptions of the messages and arguments understood by the objects adhering to this protocol. Errors generally occur because either an object does not satisfy some expected protocol, or its caller is not respecting the object’s provided protocol.

Execution is described in terms of sending messages to an object, or equivalently making requests of an object. A message is composed of the selector and the arguments. This causes the object’s class to search for a method of that name and then execute it. It creates the appearance of asking an object to do perform some action.

### 2.1.2 Program Structuring Support

Smalltalk is class-based, meaning that objects are instances of classes. Classes define the shape of their instances (the number of instance variables, or inst-vars) in addition to their behaviour (through the methods available). Instances of a class are created by sending `#new` to the class. Classes themselves are instances of *Metaclass*.

While classes and methods are the normal structuring mechanisms, most Smalltalks further support three other structuring techniques. While having no formal use in the language, they can be used for documentational and organizational purposes, and are all exploited for structuring concerns. Related methods can be grouped into categories, serving such uses as identifying methods implementing a particular protocol or serving some common purpose. Related classes can similarly be grouped through a form of class categorization, independent of their inheritance structure; the degree of support for this varies across the different Smalltalk implementations. Lastly, most environments support some management of class extensions, whereby a class can be extended with new methods. This ability, known as open classes and discussed further in Section 2.2.1, is usually used to adapt an object to support unanticipated polymorphism by adding methods to bring it into conformance with some other protocol.

### 2.1.3 The Programming Environment

While an interesting language in its own right, perhaps more interesting is the Smalltalk development environment. Smalltalk is typically programmed in very dynamic and incremental programming environments, themselves written in Smalltalk [16]. All code is interpreted by a virtual machine (VM), including the development environment. The running VM’s data/object-space is called the *image*, and contains all the objects in memory, includ-
ing the source code and running instances (themselves first-class objects). The image acts somewhat similarly to a database. Browsers provide different views onto this ‘database,’ such as for a single class, a class hierarchy, or all methods sending a particular selector.

The environment is live, meaning that any change is immediately reflected in the actual running system. Methods are compiled and bound on save; classes are created upon save too. There is no separate compile cycle, and thus the system is always in a ready-to-run state (though this differs from being tested). We describe each of these programmer-initiated operations, such as a save or deletion, as a change. These changes are scarcely perceptible: programmers rarely realize that compilation is occurring.

2.1.4 Incremental Programming
Smalltalk lends itself to incremental programming, excelling at quickly prototyping systems as only the parts of interest in a program actually need be implemented. Methods can be defined in any order, and arbitrary messages can be sent to any object, even should no method of that name exist in the system. Individual methods or classes can be changed without causing system-wide typing errors. Run-time errors are manifested as ‘walkbacks’ in the debugger, and can often be fixed on the fly. Since the development environment is just another application running in the image, it can be easily modified and added to.

2.1.5 Reflective
Smalltalk is a reflective language with a comprehensive metaobject protocol (MOP) [41]. Classes can be looked up in the system at run-time, as can methods within a class. Methods can be dynamically installed and removed, and the presence of the compiler allows methods to be easily generated and compiled on-the-fly. An object’s shape can be queried, and its class hierarchy examined programmatically. New classes can be created at runtime, and an object’s class can even be changed on the fly. And because these reflective capabilities are cheap, this object introspection is exploited by the environment through relatively sophisticated browsers and querying mechanisms. Indeed, most browsers are simply wrappers on top of the Smalltalk MOP.

Indeed, these capabilities are exploited in the browser extensions used as a case-study in Section 5.1.3.

2.1.6 Evaluation Criteria for AOP Solutions for Smalltalk
It is the environment that provides Smalltalk’s strengths, and any AOP solution must not only preserve them, but embrace them. These qualities can be summarized as follows, along with the constraints or requirements they impose for any AOP solution.
Immediate: The language extensions must preserve the immediacy, or liveness, expected of the environment: changes are immediately effected. Saving a class definition causes the class to be brought into existence, or modified to suit the new definition; saving a method causes its definition to take immediate effect. New language elements should not be any different: any changes should be reflected immediately in the system, requiring no further explicit compilation.

Incremental: The environment must continue to promote incremental development, that changes to program elements can occur in any order and at any time. Modifications should be idempotent, with the system automatically rebinding and re-evaluating the appropriate elements of the image as necessary. Any additional dependencies other than those imposed by the environment, namely that a class must exist for a method to be defined on it, should be minimized.

Reflective: The tools must support reflective facilities to facilitate building and enhancing tools. These must be read/write, so as to be able to define new elements as well as for queries.

Imperceptible: Operations should be completed in short order; the programmer should not be aware of any separate compilation process. Any results, such as generated code, should be efficient and not introduce noticable lag.

These properties actually apply to any tool for the Smalltalk environment: any tool must be fully integrated with the environment, and not be seen as a mere pre-processor. Failure to meet these criteria will likely lead to the tool’s rejection, as it is unlikely that any tool can be good enough to warrant sacrificing them.

2.2 Prior Work Leading to AOP

There have been several approaches leading to AOP. This section briefly describes the work arising from the research into meta-level programming leading to the form of AOP in AspectJ.

2.2.1 Open Classes

The concept of ‘open classes’ was first introduced through Flavors [9]. It refers to a system where new methods can be dynamically added to classes without modifying the original class definition [10]. This allows ‘adjusting’ classes to support previously unanticipated behaviours or uses, useful when integrating different packages as classes can be made polymorphically compatible. Alternative solutions require ugly contortions such as object-wrapping, which usually cause other problems, such as maintaining object-identity.
Smalltalk has open classes, allowing arbitrary methods to be added, removed or replaced at will without recompilation of the class or any change to the class identity. C++ and Java do not have open classes.

Open classes can suffer from name collisions, when one tries to introduce already-defined methods on a class. These collisions can be avoided by prefixing conventions, which work well in practice. While prefixing may seem distasteful, uncontrolled extending can otherwise lead to chaos, especially when different developers choose different semantics for similarly- or identically-named methods.

2.2.2 **Generic Functions in the Common Lisp Object System**

Generic functions, originating from CommonLoops [4], look like and can be treated as Lisp functions. But 'gfuns' are different, and are perhaps best thought of as 'place-holder' functions. Methods are associated with a gfun, but are only be executed if the arguments are of certain types. These methods can be further qualified to execute before or after the primary method. These qualifiers allow attaching behaviour to methods. But their definition must specify the specific gfun to be combined with, thus there is little detachment between the behaviour and where it applies.

2.2.3 **Programming the Meta Level**

Some of the concerns that are hardest to isolate crosscut the entire system, such as distribution or persistence. It was surmised that these can perhaps be best accommodated by modifying the actual language, rather than the program. This is programming the metalevel of the language: changing how the language functions. For example, an object can be made asynchronous by changing how it receives requests.

Much effort has been expended investigating this area, and more on its management in a principled fashion (e.g. [27, 34]). But while powerful, meta-level programming is hard. It is difficult to express changes as modifications to the language semantics, and to reason about the implications of a change across a system. By changing the meaning of the language, the program text may no longer even approximate the actual execution: what is seen may not be what is executed. This also complicates building tools to help understand the changes.

2.3 **Aspect-Oriented Programming**

Aspect-Oriented Programming (AOP) proposes introducing new language constructs to provide principled ways for separating and capturing crosscutting concerns as part of the
language [30]. This aids in elegantly modularizing the crosscutting concerns left uncaptured by other mechanisms in the language.

There have been two fundamental approaches to AOP: through domain-specific languages such as AML [25], D [33] and RG [35]; or through general-purpose aspect languages. Recent research has focused on the latter by retrofitting existing languages with aspect-oriented extensions. This provides new tools to the "programmers in the trenches" while preserving their investment in existing code-bases. Several such retrofits currently exist including AspectJ, Hyper/J [46, 38], Composition Filters [3], and Demeter [32]. AspectJ is discussed further in the next section.

The central elements of AOP languages are:

**Join Point:** Join points are points in the program, providing the frame of reference for describing the program. These are 'the places where aspect code interacts with the rest of the system' [14], and are not required to correspond to language elements.

**Identification of Some Join Points:** This provides an explicit structure for describing, manipulating and understanding some portion of the program.

**Encoding the Behaviour to be Effected at the Join Points:** This specifies the manner in which the aspect program affects the behaviour of the program.

How they are provided or presented in languages varies widely.

### 2.3.1 AOP's Place in the World

AOP is designed to complement, not supplant, other program structuring methods. AOP provides language support to provide alternative modularizations, done by other means due to the lack of this support. These other mechanisms (callbacks, event systems, etc.) all require some modifications to provide hooks to the original system.

It is important to note that AOP does not claim to replace refactoring to best use AOP. Refactoring is sometimes required to adequately isolate the conditions of interest. In the brief simulation extract from Section 1.2, the completion of this method does not necessarily indicate successful scheduling of a node. To accurately capture only this requires that the main body of the method, the part after having filtered out disabled nodes, be split into a separate method, and this method cited where needed.

### 2.4 The AspectJ Model of Aspect-Oriented Programming

AspectJ is a conservative attempt to extend Java with AOP constructs. It has evolved to become a general-purpose, aspect-language [1, 29], a superset of the Java language. AspectJ
models AOP as a three-part declarative representation of annotating a program’s execution. It specifically implements these as the following:

**Pointcuts:** Join points are identified using a *pointcut*, matching some set of properties. Pointcuts are built by composing a small set of primitive pointcuts using standard set operations (intersection, union, and complement). Pointcuts can be defined and named on classes and aspects much like methods. This naming allows separating their definitions from their use, promoting information hiding [39].

**Advice:** Advice specifies code, called the advice body, to be executed when execution reaches the join points identified by a pointcut. The code can either be executed *before* the point occurs, or *after* it has occurred. A special combination, *around*, allows modifying the context at the point, even replacing it by not continuing to the point at all. Selected parts of the execution context at the join point can be exposed to, and even modified by, advice. Advice is defined only on aspects.

**Aspects:** Aspects are like classes, but make explicit the state of a crosscutting concern. Aspects are never explicitly instantiated, but are rather created upon the satisfaction of some per-aspect condition, and last for the duration of this condition.

AspectJ features a rich join point model, identifying join points such as requests to execute a method on an object, the actual execution of a method, calls to a constructor, gets and sets of a field, or static initialization, amongst others. Its primitive pointcuts allow identifying join points using a variety of characteristics, such as by type, method name, the caller’s type.

### 2.4.1 Interface

AspectJ is, like Java, a declarative approach with limited reflection. While some language elements are made explicit in the environment, there is no support for dynamic construction of the language constructs. Advice, for example, cannot be constructed and then applied at run-time.

### 2.4.2 Implementation: Static Weaving

In terms of implementation, the ajc implementation of AspectJ uses a compile-time two-step *weaving* process. The first step involves identifying source locations corresponding to the join points identified by the pointcuts. But a code location may only partially correspond to a join point as the join point may be dependent on its execution context, as might happen with *cflow()*, a primitive identifying conditions in the control flow. The second step of the static weaving requires performing code transformations of these locations to first perform
residual tests, testing that the dynamic elements of the join point are matched, and then call the applicable advice.

2.5 Previous AOP-like Solutions for Smalltalk

This section describes some of the other major technologies used for separating concerns in Smalltalk.

2.5.1 Improved Structuring of Smalltalk Applications with ENVY/Manager

Traditional Smalltalk environments provide little further application structuring support: there are no structures for capturing a group of classes and introduced methods that form a concern. Related classes can be categorized with a common name. But newly-introduced methods to a class can only be categorized with methods within the class, and thus appear categorized with the class and not the concern. It is difficult to consider the code related to a particular concern in isolation.

ENVY/Manager [37] is a sophisticated configuration management and version-control tool for Smalltalk providing formal support for grouping related classes and methods using structures called Applications. An Application can not only define new classes, it can extend existing classes with new methods; these methods remain associated with their defining application, and not the extended-class' defining application. Figure 2.1 shows a browser window on the ApAdvice class hierarchy: the first of the upper panes shows the class hierarchy, and the second the applications which extend the class. This shows that ApAdvice has four extensions in addition to its defining application. Method collisions can occur, but are detected at load-time and left to the programmer to resolve.

ENVY/Manager's management structures are reified in the environment, complementing Smalltalk's reflection capabilities. This allows queries such as determining the classes defined or extended by a particular Application, finding a class' defining Application, or an Application's prerequisite applications. A call-back mechanism publishes changes to the system, such as method or class additions, deletions or modifications. There is also support for start-up/initialization and tear-down code when loading or unloading applications from the image.

In fact, ENVY/Manager has effectively provided a subset of AOP-like constructs for Smalltalk for one and a half decades [36]. This remains only a subset, as ENVY/Manager does not provide any method composition facilities. But these could be achieved in some form, perhaps, by combining the per-application start-up/tear-down code and callbacks with method-wrapping (see Section 2.5.2). No such use has been documented however.
Figure 2.1: A browser showing the many extensions of the class `ApAdvice` by multiple applications (second of the upper panes). The third upper pane shows the method categories for this class, and the fourth the methods belonging to the selected categories.
2.5.2 MethodWrappers

MethodWrappers [7] are a mechanism for adding behaviour to individual Smalltalk methods. They support executing some behaviour before and/or after the method has been executed, acting somewhat similar to CLOS’ :before, :after, and :around qualifiers with generic functions (discussed in Section 2.2.2). The implementors examined many different implementation strategies, including source-code modifications (the approach used by Apostle), proxying, class-replacement, and byte-code modifications, before settling on actual method replacement.

MethodWrappers work by replacing the actual CompiledMethod (the native Smalltalk representation of a method) with the wrapper, itself a form of CompiledMethod. This causes the wrapper-method to be executed when the wrapped selector is received by an instance, effectively intercepting the method. The wrapper can then choose to do something before or after the wrapped method, or not execute it at all.

MethodWrappers are defined as subclasses MethodWrapper, and then explicitly instantiated and installed for each particular method. Such code might look like the following:

```smalltalk
| methodWrapper |
methodWrapper := methodWrapperClass on: #printString inClass: Foo
methodWrapper install.
```

Wrappers are removed by sending #uninstall to the wrapper instance. This has a very low cost, both in creation and execution, as the code is compiled at definition-time, and only requires simple instantiation at install-time.

The authors present several interesting applications of MethodWrappers, including pre- and post-conditions browsers, automatic sequence diagramming, and dynamic class-coupling diagramming used with vanilla Smalltalk programs. All illustrate the usefulness of MethodWrappers: none would be possible without significant intrusive changes to application code. These MethodWrapper examples work well as have a common style of crosscutting: they have few types of method wrapper, and those are relatively simple.

But MethodWrappers do not do as well when dealing with small, non-pervasive, sets of changes, especially ones requiring many types of wrapper. There seems to be little support for managing groups of wrappers, perhaps due to their being considered only a mechanism: there is no knowledge maintained about where wrappers have been applied to, nor for determining whether a particular method had been previously wrapped. And as there is no support for combining or composing wrappers, some significant amounts of infrastructure would need to be built to use MethodWrappers for capturing multiple smaller crosscutting concerns or more complex systems, as managing many different types of wrapper applied to many different places is difficult. Such infrastructure appears to have been built for the pre-/post-conditions example.

MethodWrappers do illustrate the utility of a programmatic interface/MOP for dy-
namically applying cross-cutting behaviours. Oftentimes the actual parameters of the problem solution may not be known until run-time, long after the concern in question has been written.

2.6 Prior AOP Solutions for Smalltalk

This section describes four other AOP solutions for Smalltalk.

2.6.1 Andrew

Andrew [20] is a recent contribution producing an AspectJ-style AOP language for Smalltalk. Andrew replaces the AspectJ-style pointcut language with a logic meta programming language, extended with certain predicates so as to form a new pointcut language. By using a logic meta programming language, the pointcut language is easily extended with new pointcut types.

Andrew's implementation follows a static-compilation model, much like Apostle, where join points are mapped to feasible locations in the code. But it requires an explicit weaving step, and as such does not support incremental compilation.

2.6.2 AOP/ST

AOP/ST [5, 6] is a straight-forward port of an older variant of AspectJ to Smalltalk. This variant provided support for building domain-specific languages. Part of this porting effort involved implementing COOL [33], a language for expressing concurrency, as well as an execution tracing language. AOP/ST presents a pure-Smalltalk binding, meaning that it introduces no special language constructs, instead reducing all its capabilities to traditional Smalltalk messages. It also requires a separate compilation/weaving step; there is no support for immediacy.

The implementation uses 'lightweight classes': for every affected class, AOP/ST builds a special instance of Behavior with appropriately-overidden methods, and having the exact shape of the class being replaced. All instances of the original class then have their class slots bashed, or ungracefully replaced, with this new lightweight class. This means that method-dispatch first queries the lightweight class for the method implementation before proceeding to the original class, thus allowing methods to be easily intercepted. While a somewhat expensive operation to install the lightweight class, subsequent method lookups are fast.
2.6.3 Pryor and Bastán’s Metaobject Architecture for AOP

Very little evidence of this system exists beyond a single workshop paper [40]. The system follows the dynamic-association model presented by AspectJ version 0.5, another older variant where objects are explicitly associated with an aspect.

The system is implemented using the Luther Metaobject architecture [8] for Smalltalk. Aspects are represented by metaobjects, inheriting from the framework’s MetaObjectClass. Messages sent to an aspect-managed object (i.e. one that has been explicitly associated with an aspect) are intercepted and redirected to the AspectManager. This manager identifies the relevant aspects for the object, and sends a #handleBeforeMsg: to each in turn. After executing the actual method, the AspectManager then sends #handleAfterMsg: to the respective aspects. The aspects are responsible for any further filtering for advice, such as by their selectors.

The interface, as near as can be determined, is very mechanistic: some code somewhere must associate the requisite aspects with the object instances. But the implementation is conceptually attractive.

2.6.4 AspectS for Squeak

AspectS [22] is a research project investigating the requirements for of adding AOP support to the Smalltalk-variant Squeak [45]. AspectS loosely follows the current AspectJ 1.0 model, but with a limited join point model identifying only method executions. It is implemented using MethodWrappers (see Section 2.5.2), and could be seen as a wrapper management shell.

The initial investigation of language extensions was abandoned to focus on an metaobject protocol (MOP). Advice is defined on subclasses of AsAspect through specially identified methods prefixed with #advice. This prefixing is necessary as the advice is built using a Smalltalk expression and thus must be explicitly executed to be brought into effect. The following is an example taken from the AspectS documentation.

```plaintext
adviceMouseEnter

^AsBeforeAfterAdvice new
  qualifier: (AsAdviceQualifier major: #receiverGeneral minor: nil);
  joinPoints:
    ((Smalltalk allClassesImplementing: #mouseEnter:)
      select: [:each | each includesBehavior: Morph]
      thenCollect: [:each |
        AsJoinPoint new
          targetClass: each;
          targetSelector: #mouseEnter:]);
  beforeBlock: [:receiver :arguments :aspect :client |
    self
      showHeader: '>> MouseENTER >>';
    receiver: receiver
```
AspectS has no equivalent to pointcut designators. Join points are instead explicitly identified by the programmer using Smalltalk's reflective capabilities (lines 6–11). Advice-bodies are provided as blocks (lines 12–16), which requires careful use due to Squeak's lack of full-closure in blocks. All method arguments are presented unfiltered to the advice body.

'Advice qualifiers' (line 4) dictate the applicability of the advice. #receiverGeneral means the advice applies to any instances of the identified classes; #receiverSpecific causes the advice to only pertain to some explicitly identified set of instances. Other qualifiers allow being sender-specific; minor qualifiers add cflow-style semantics.

While its use of Smalltalk code for identifying join points is interesting, removing the need for programmers to learn yet another mini-language, it has disadvantages.

AspectS cannot support incremental development as there is no separation between the identification of join points and the advice: there is no equivalent to pointcuts. Advice is applied to an explicitly identified list of join points, usually identified using Smalltalk's reflection facilities. These join points are then fixed once the advice has been put into effect, and there is no manner for the system to determine whether new method definitions should be matched by this advice.

The lack of pointcuts can also result in suboptimal querying, as the programmer may not choose a good ordering in the query. This is doubly important when composing a set of filtering operations. The results can sometimes be obtained more quickly by reordering the traversal of the image, as done in databases. This requires some structure for the system to be able to perform its optimization.

Presenting all the arguments from the method can be problematic. Consider where the advice block is applicable for two different methods with different argument orderings. The advice must spend time determining which of the arguments is the one of interest.

2.7 Summary

This chapter discussed some of the important stylistic and cultural differences amongst Smalltalk programmers and environments. It also briefly discussed some of the work leading to AOP and AspectJ. Section 2.1.6 defined a set of criteria for evaluating Smalltalk AOP solutions, and briefly analyzed some existing AOP solutions for Smalltalk according to these criteria.
Chapter 3

Modelling Aspect-Oriented Programming In And For Smalltalk

This chapter presents Apostle, an aspect-oriented extension of the Smalltalk language. It leverages much of its model from AspectJ, discussed in Section 2.4, but remains upward compatible with the Smalltalk language. Apostle has two facets: a declarative style, suitable for use within the browser environment, and a metaobject protocol (MOP) suitable for dynamic program construction. Section 3.1 discusses some of the constraints for the language: these dictate some of how the language appearance and implementation.

Section 3.2 attempts to convey what it is like to program in Apostle through the use of examples; a more formal specification of Apostle is found in Appendix B. Section 3.3 discusses the rationales for the decisions made in the design of the language. The actual implementation strategy is left to Chapter 4.

3.1 Design Constraints

Section 2.1.6 summarized the key properties that must be maintained by Apostle. These criteria lead to the following requirements of the language:

**Immediate:** Section 2.1.6 specifies that any changes must be immediately reflected in the environment. Then changing a pointcut or advice definition must cause their new definitions to take immediate effect. These new language elements should be presented in browsers similar to methods, and be able to be placed under version control.

This requires these elements have a distinguishable, declarative form. This allows the compiler to immediately discern the type of a code-fragment, whether it be a method, advice or pointcut, and handle it appropriately.

This cannot be accomplished when representing advice or pointcuts as specialized Smalltalk methods, as they can only be effected by executing the method. This is
never done explicitly by the environment; to do so would require some form of marking the method as such. But these markings, through naming conventions or the like, while workable, are inelegant.

**Incremental:** To preserve incrementality, the order in defining a method and a pointcut identifying that method must not matter. All changes are applied immediately and rationalized within the environment. The same effect must occur: the method will be targeted with the proper advice.

**Reflective:** There must be some programmatic form for accessing and controlling the language.

**Imperceptible:** Sufficient information must be maintained so any change can be quickly fulfilled.

### 3.2 Language Fundamentals

Apostle is closely modelled on AspectJ, featuring pointcut designators, advice and aspects, thereby leveraging the knowledge embodied in AspectJ's design and evolution. **Pointcuts** identify some set of join points of the program. **Advice** specifies code to be executed upon reaching these join points. And **aspects** serve as a repository for the state of a crosscutting concern.

#### 3.2.1 Join Point Model

Apostle has a much simpler join point model than AspectJ due to Smalltalk's simplicity as compared to Java. Only two types of join point are identifiable: the reception of a message by an object, and the execution of the methods corresponding to that message.

There are no separate join points describing constructors and class-initializers: these are implemented as methods on the class and cannot be distinguished from other methods, and thus cannot be specifically targeted. But being just methods, these join points can be captured by treating them as normal methods. Nor does Apostle provide distinct get-instance-variable or set-instance-variable join points. This is considered acceptable since the instance variables of a Smalltalk object cannot be directly accessed outside of the object except through the use of accessor methods, which can be captured.

Rather than present a dry formal description of the language, we motivate it through selected examples, either simple ditties, or extracts from the case studies from Section 5.1.1.
3.2.2 A Simple Example: Expressing the Example from the Introduction

This first example expresses the window-updating example from Section 1.2 using Apostle. It demonstrates identifying join points using a pointcut, and affecting the program using advice.

To summarize, the simulator is loosely architected as a model-view-controller system where a \texttt{PercolationSimulator} is viewed by a \texttt{SimulationWindow}, which also serves as the controller. The window is updated as nodes are added, become scheduled, or are fired during the simulation. The situation examined code to update a simulator's view whenever a new node has been scheduled. The programmer had isolated the necessary places for such notification to one method, \texttt{PercolationSimulator\#schedule:}. But the implementation required appending a seemingly-extraneous line to the end of the method.

This would ideally be expressed as 'having scheduled a node, update the view.' This can be expressed with the following advice using Apostle:

\begin{verbatim}
1  after <kindOf(PercolationSimulation) & receptions(#schedule: node)>
2  window notNil ifTrue: [window drawNode: node]
\end{verbatim}

The first word on line 1, after, specifies that this advice is \textit{after} advice, to be executed after any of its identified join points have occurred. The text between the angle brackets, '&', specifies the \textit{pointcut}, an expression used to identify relevant join points. The remainder of the advice defines the \textit{advice-body}, the actual code to execute having reached the applicable join points.

Pointcuts are expressed in similar manner as in AspectJ, but use more Smalltalk-like set operators for intersection (`&`) and union (`|`); negation remains the same (`|`). This particular pointcut matches the join points where an object of type \texttt{PercolationSimulator} actually executes the method named \texttt{#schedule:}. Note that the \texttt{executions()} pointcut names the argument to \texttt{#schedule:}. This allows the advice-body to refer to that argument by name.

3.2.3 Naming Pointcuts

Pointcuts are also first-class program elements, able to be defined and named on both aspects and classes, resembling class methods. This can be used to dissociate the definition from its use. For example, if there were multiple ways of scheduling a node, then the simulator class could identify the as a named pointcut as follows:

\begin{verbatim}
pointcut scheduledNode(simulator, node):
  kindOf(PercolationSimulator simulator) & executions(schedule: node)
\end{verbatim}

The advice would then be rewritten as:

\begin{verbatim}
after<PercolationSimulator.scheduledNode(simulator, node)>
  (window notNil and: [window simulator = simulator])
  ifTrue: [window drawNode: node]
\end{verbatim}
3.2.4 Storage of State Related to an Aspect

Crosscutting concerns can carry state. This is stored in aspect instances. An aspect is like a class in that it gathers the behaviour and state of the crosscutting concern.

Continuing the preceding example, window is state associated with this crosscutting concern that can be maintained through an aspect. Aspects are created in a similar manner as classes, by sending a message to their super-aspect:

1. ApAspect subaspect: # SimNotification
2. instanceVariableNames: 'window '
3. classVariableNames: ''
4. poolDictionaries: ''

This defines an aspect named SimNotification, a subaspect of ApAspect; all aspects ultimately inherit from this aspect. SimNotification is a singleton aspect, meaning that only one instance ever exists in the system, like the Singleton pattern [15]. There are two other types of aspect, which are described in Appendix B.1.4.

Advice is defined on aspects and can access the state of their associated aspect instance; hence the advice above can use window, providing it is defined on this aspect.

The window must somehow be set. This is done through advice.

after<kindOf(SimulationWindow view) & receptions(simulator:)>

window := view

Whenever a simulation window (the view) sets its simulator, the aspect stores the window.

This is not an ideal solution: because it uses a singleton aspect, there can be at most one such pairing running at a time. Please refer to Section 5.1.1 for a more realistic solution supporting multiple views on a simulator, in addition to supporting multiple simulators.

3.2.5 Other Aspect and Advice Types

Continuing the simulation example, we next attempt some window-updating optimizations. This uses a different type of aspect, and a different type of advice. An aspect's type can differ from its superaspect's.

The window-updating attempts to be intelligent, drawing only the nodes that have changed. But the window must sometimes be completely redrawn as its current view has become out-of-date (e.g., the simulation has scrolled off the screen). Any node-updates that occur during a redraw should be discarded. We implement this optimization as follows.

First define a new aspect, called SimDrawingOptimization:

1. ApAspect subaspect: # SimDrawingOptimization
2. perObject: 'kindOf(SimulationWindow )'
3. instanceVariableNames: 'redrawing '
4. classVariableNames: ''
5. poolDictionaries: ''
Line 2 specifies this aspect to be a per-object aspect. Per-object aspects are instantiated on a one-to-one basis with any objects matching the provided pointcut, and exist for the lifetime of their corresponding object. A per-object aspect is necessary for this example as each simulation-window redraws independently.

Line 3 declares that instances of this aspect are to have an variable named **redrawing**; this is a flag indicating whether a redraw is in progress. This is true if a redraw is in progress, and false otherwise.

**Initializing New Aspect Instances.**

As aspects cannot be explicitly created with **#new**, Apostle causes all new aspect instances to be sent **initialize** upon creation. We override **initialize** to initialize the **redrawing** flag to false.

```plaintext
    initialize
    redrawing := false.
    ^super initialize
```

**initialize** is a normal method: aspects can define methods in addition to pointcuts and advice.

**Defining Advice.**

Having defined an aspect, we proceed to define the advice implementing the drawing-optimizations. Advice can only be defined on aspects.

```plaintext
    around<kindOf(SimulationWindow) & receptions(#redrawGraph)>
    redrawing == true ifTrue: [^self].
    redrawing := true.
    ^(PROCEED) ensure: [redrawing := false]
    around<kindOf(SimulationWindow) & receptions(#drawNode:)
    redrawing == true ifTrue: [^self].
    ^(PROCEED)
```

Around advice is different from before and after advice in that it can directly affect the join point whereas having separate before and after advice cannot. Around advice can change the execution context provided to the join point, or change the value returned by the join point. The actual execution point is continued using the **PROCEED** keyword. The result of the **PROCEED** is the actual result of the execution point; around advice can change this by returning a different value. In fact, around advice can even prevent the execution of the join point by omitting the **PROCEED** entirely. We use around advice for exactly this reason, as it allows the advice to skip the redraw should a redraw be in progress.
3.2.6 The Execution Context of Advice

Executing advice is provided a special object named thisJoinPoint encapsulating its current execution location. It can be queried for the object’s type, and the selector name.

3.2.7 Affecting The Execution Context of a Join Point

It is sometimes desirable to change the context provided to the original join point, such as providing different arguments. This is accomplished using around advice, name binding, and the PROCEEDUSING keyword. This is illustrated this with the following example that supplements Smalltalk’s reflective facilities.

The instance variables of an object can be obtained with #instVarAt:; this is handy when a class does not define accessors for all of its variables. But #instVarAt: takes the variable’s index, and not its name. These indices may change when the shape of an object is changed, as might happen when adding or removing instance variables. But the names of the instance variables are easily obtained and mapped to their appropriate indices. We supplement #instVarAt: and #instVarAt:put: to take both an index or a variable name:

```
around executions (#instVarAt : key) | executions (#instVarAt : key put:)
```

```lisp
| index |
key isInteger ifTrue: [^PROCEED].
index := self class allInstVarNames indexOf: key ifAbsent: [^nil].
^PROCEEDUSING key: index
```

This advice uses name-capturing to selectively expose parts of the execution context at the join point to the advice. The executions() clauses bind the name key to be the index argument; this argument will be provided to the advice-body when executed. Many primitive pointcuts provide such capturing (see Appendix B.1.1).

PROCEEDUSING is a variant of PROCEED that allows some limited changing of the context passed on to the join point. This is sent a special keyword-selector whose keywords match the names of the formally-bound arguments. Each specified name’s value is overridden by the provided argument. The keywords can be specified in any order; names not specified will continue with their original values. Currently only those names corresponding to the current method arguments can be overridden; overridden non-arguments fail silently and retain their original value.

3.2.8 Summary

This has provided a simple walk-through of most of Apostle’s features. There is an additional aspect-type, per-cflow, and additional primitive pointcut designators. These are described in more detail in Appendix B.
3.3 Discussion and Justification of Language Design

This section discusses the rationale behind Apostle’s appearance, first presenting broad arguments to justify having a separate language for Apostle in Section 3.3.1, followed by Sections 3.3.2 and 3.3.3 presenting more detailed arguments about the form of pointcuts.

3.3.1 The Necessity for a Declarative Language

Apostle is not the first tool to propose new extensions to the Smalltalk language. Iona Orbix for Smalltalk, based on Hewlett-Packard’s Distributed Smalltalk, presents its CORBA IDL definitions in the normal Smalltalk browsers [44]. SmalltalkAgents™ has been evolving the Smalltalk-80 syntax for over a decade. Squeak has introduced new constructs to the language, such as the brace notation for building arrays of objects [18]. Each of these extensions have been considered warranted as the existing language did not provide enough support to clearly express the intent. We similarly believe the declarative Apostle language better expresses the intent of the advice than any pure Smalltalk form for the following reasons:

Conciseness and Clarity. Contrast the following piece of advice, expressed using the Apostle language:

```plaintext
before<kindOf(Collection) & receptions(at: key put: value)
   & cflow(recordChanges())>

aspect recordChange: self at: key put: value
```

with the following Smalltalk fragment using the Apostle MOP (Appendix B.2):

```plaintext
(advice := ApAdvice type: ApBeforeAdviceType)
pointcut:
   {{(ApKindOfPrimitivePointcut typeName: 'Collection')
      & (ApReceptionsPrimitivePointcut selector: 'at: key put: value')
      & (ApCflowPrimitivePointcut pointcut:
         (ApNamedPointcutReference referencing: 'recordChanges'))};
snippet: 'aspect recordChange: self at: key put: value';
apply
```

It is difficult to define any quantitative reason why one is better than the other, beyond that the equivalent MOP code is larger. But qualitatively, the declarative form contains only the necessary pieces necessary: it is trivial to discern the advice type (before), the pointcut, and the advice-body; this is not true of the MOP code. The MOP variant also includes mechanical details that distract from the purpose of the code.

Comprehensibility. Following on the previous example, attempting to capture the join point description in Smalltalk requires comparatively cumbersome verbiage. The language
was not designed to express concepts such as describing moments in the execution context. This is expanded upon in Section 3.3.3.

**Distinguishability.** There is little confusion whether a code fragment is advice, a pointcut or a method or MOP program. This is important for both users and the environment. Without a distinct format, the environment cannot know whether the method is a ‘normal’ Smalltalk method or a special MOP-building method, save through special naming conventions, as done in AspectS. This is essential for immediacy, as explained in Section 3.1

**Freedom from Pre-conceptions and Inhibitions.** A non-Smalltalk syntax frees the language from many strictures: there are no preconceived meanings associated to elements of the representation, nor any expectations to provide other Smalltalk constructs, and any inhibitions in adding non-Smalltalk constructs are removed.

**Summary** In short, we feel it is better to use a mini-language that fully and compactly supports the necessary structures and concepts rather than force our needs to fit some artificial semblance of the language. It removes any need to ascribe special meaning to certain expressions in the language, such as having specially prefixed method-names. It presents an unambiguous description, exploitable by tools. We believe the purely-Smalltalk solution suffers in all regards.

### 3.3.2 Justification for Expressing Pointcuts as Predicates

Apostle’s pointcuts are deliberately considered as predicates for performance reasons. Being purely functional, the components of a pointcut commute, allowing them to be reordered by Apostle with no semantic difference during compilation. Consider that VisualAge for Smalltalk (VAST, a popular development environment) ships with over 3000 classes in its standard image. The Apostle weaver must, on any pointcut use, identify all relevant elements within the image. A pointcut re-ordering that eliminates large swathes of the image from consideration will have a dramatic impact on performance.

This point is illustrated with the following pointcut:

```plaintext
receptions(#at:) & kindOf(Array)
```

Identifying all the possible reception points of `#at:` requires querying the methods of all classes of the system, which is a much more expensive operation than finding a particular class and traversing its hierarchy. `kindOf()` uses selection criteria that naturally aligns with the existing class-graph structure, enabling it to dramatically reduce the search-space.
3.3.3 Exploring a More Smalltalk-like Pointcut Language

Justifying Apostle’s distinctly non-Smalltalk pointcut syntax is a seemingly weaker case. While we did attempt to initially express pointcuts in a more Smalltalk-ish manner, it clashed with our requirements that pointcuts be treated as predicates.

Pointcuts being predicates requires their representation have no implied no order of evaluation. This is not adequately suggested with Smalltalk’s typical short-circuiting boolean operations #and: and #or:. Replaced with the non-short-circuiting operators, ‘&’ and ‘|’, might produce something similar to:

\[
(s\text{elf } \text{receptions: } \text{#at:put:}) \& (s\text{elf } \text{kindOf: } \text{Collection}) \& \\
(s\text{elf } \text{kindOf: } \text{Array}) \text{ not}
\]

But it is not clear what ‘self’ represents.

An alternative formulation considered the pointcut declaration as a construction of a query, where the receivers are classes representing the primitive pointcut designators. For example:

\[
(\text{Receptions selector: } \text{#at:put:}) \& (\text{KindOf type: } \text{Collection}) \& \\
(\text{KindOf type: } \text{Array}) \text{ not}
\]

While less compact than the AspectJ-style functional representation, this closely resembles the protocols supported by the Apostle MOP.

But this solution suffers when exposing selected elements of the context at the join point. Consider attempting to capture the arguments of #at:put:. A naive attempt:

\[
\text{Receptions selector: } \text{#at: key put: value}
\]

results in ambiguous code. As a Smalltalk expression, this sends #selector:put: to Receptions, having sent #key to the symbol at:. While this can be disambiguated using special notation for selectors with named arguments, such as:

\[
\text{Receptions selector: } \#[\text{at: key put: value}]
\]

this introduces new and decidedly non-Smalltalk syntax: the exact problem we were attempting to avoid. This does not address other forms of context naming, such as occurs with KindOf; these other situations would require new syntax too.

While a more Smalltalk-like syntax seem appealing, the ambiguities and difficulties in fitting the language to such a style outweigh its perceived advantages. We kept the more compact and functional notation used in AspectJ.

3.4 Summary

This chapter has presented the Apostle language, and much of the reasoning behind its appearance. Although based on AspectJ, it differs in the appearance of many of the constructs either because of Smalltalk’s lack of static typing, or the need to differentiate its constructs from Smalltalk’s existing constructs.
Chapter 4

Implementation of Apostle

The implementation of AOP languages requires an operation known as weaving, which involves resolving the crosscutting so as to ensure that advice is called appropriately throughout the code. Most implementations do compile-time, whole-program weaving where the entire program is re-woven in a static process, first weaving the aspect code into the program by mapping join points to actual code locations, and then compiling this resulting aspect-free code.

With Smalltalk being a run-time interpreted language, we supposed that an AOP implementation similarly required an interpreted process, where the interpreter dynamically determining the aspect code at run-time. Our plan was to modify the method dispatch loop of the freely-available Squeak [45] virtual machine (VM) to check for applicable advice before and after each method send.

But frustrations with the VM caused us to pursue an incremental static weaving solution, whose re-weaving performance proportional to the change in the program. Static weaving can be made to appear dynamic providing it is performed immediately and sufficient information is kept to maintain consistency with future updates.

This solution is discussed in further detail in the remainder of this chapter.

4.1 Overview

Apostle is implemented using an incremental source to source strategy. The overall strategy is to weave the advice and source into an equivalent pure-Smalltalk result, called the target model (Figure 4.1). This means that Apostle requires no modifications to the underlying virtual machine. Our incremental weaving can be alternatively seen as following cycle shown in Figure 4.2: any program change is examined to determine its impact, resulting in regenerating certain pieces in the target model. The Apostle implementation has concentrated on Steps 2 and 3, emphasizing the correctness of the implementation rather than
Figure 4.1: The implementation structure of Apostle.

Figure 4.2: The steps in an incremental weaver.
the speed of the generated code. As such many implementation strategies are non-optimal, using simple approaches.

This section proceeds as follows. Section 4.2 details the Apostle target model: the result of the source-to-source transformation of Apostle programs to pure Smalltalk. Section 4.3 then examines how the various incremental program changes impact this transformed base, and how to correct for them.

### 4.2 The Apostle Target Model

Apostle uses a source-to-source compilation strategy, weaving the advice and source into an equivalent pure-Smalltalk result called the target model.

The semantics of the Apostle language requires executing advice whenever execution reaches a join point matched by its pointcut. To statically weave the program, the Apostle weaver finds the shadows of each join point, code locations possibly corresponding to a dynamic join point, and check every advice declaration to see whether its pointcut could possibly match the shadow. For each possibly-matching shadow, the weaver determines the residual work necessary to ensure the advice applies: some pointcuts require only statically available information to determine their relevant join points; others require dynamic run-time tests. Each of these shadows are then statically modified to dispatch all advice matching the join point.

For runtime efficiency, the implementation makes some assumptions which are embedded in its generated code. To accommodate incremental development, these assumptions are recorded for use in corrective actions for subsequent changes.

#### 4.2.1 Join Points and Join Point Shadows

As specified in Section 3.2.1, Apostle's join point model identifies points in the runtime object call graph, specifically method receptions and method executions. Even though join points are points in the execution, we can still identify places in the code that are "where execution will be" whenever execution reaches a join point. These places are called join point shadows. One consequence of this model is that some information associated with a join point shadow can be statically determined. Because the Apostle language only identifies join points at a method granularity, shadows always correspond to methods, and never sub-method join points.

For example, the shadows for reception and execution join points for `Point>>#x` are:

```plaintext
Point>>x: newX

<notSuperSend> ifTrue: [<reception join point shadow>].

<execution join point shadow>
```
The only difference between a method reception and execution depends on what caused the method to be executed in the first place. An join point is a reception join point if and only if the method was not executed as a result of a super-send. This can be determined by checking that:

1. the caller object is the same as the called object;
2. the caller method selector is the same as the called method selector;
3. the caller method defining class is not the same as the called method defining class.

Step 3 is essential to differentiate super-sends from recursive sends. These tests are simple to perform at run-time given that most Smalltalk VMs provide such stack-searching operations, and they are relatively cheap.

Properties Associated with Join Points

One consequence of this model is that some of the properties associated with join points are statically known at join point shadows. For example, at the shadow of execution join points for Point>>#x: we know that the name of the method is #x:. Other properties are only known dynamically, such as the arguments to the method, or whether a reception shadow actually corresponds to a reception join point. There is a well-defined distinction between the properties able to be determined statically and dynamically at a join point shadow; these are summarized in Table 4.1.

We can understand this by enhancing the shadow shown above to build actual run-time representations of the join points:

```plaintext
Point>>x: newX
```

```plaintext
| thisReceptionJoinPoint thisExecutionJoinPoint |
<notSuperSend> ifTrue: [
  thisReceptionJoinPoint := ApReceptionJoinPoint new
  receiver: self;
  receiverClass: Point;
  selector: #x:\;
  arguments: {Array with: newX};
  sender: <vmSender>;
  sendingMethod: <vmSendingMethod>].
```

---

1 Stack-searching operations take approximately 1μs on a 750MHz Pentium III with VAST 5.5.2; this compares to a method-send time of 0.119μs
<table>
<thead>
<tr>
<th>Property</th>
<th>Known Statically at Shadow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>called object</td>
<td>yes</td>
</tr>
<tr>
<td>called object class</td>
<td>yes</td>
</tr>
<tr>
<td>called method name</td>
<td>yes</td>
</tr>
<tr>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>active process</td>
<td></td>
</tr>
<tr>
<td>caller object</td>
<td></td>
</tr>
<tr>
<td>caller object class</td>
<td></td>
</tr>
<tr>
<td>calling method</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Known Statically at Shadow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>called object</td>
<td>yes</td>
</tr>
<tr>
<td>called object class</td>
<td>yes</td>
</tr>
<tr>
<td>called method name</td>
<td>yes</td>
</tr>
<tr>
<td>arguments</td>
<td></td>
</tr>
<tr>
<td>active process</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Properties associated with method reception and execution join points, with indications of those determinable statically at a shadows.

```plaintext
thisExecutionJoinPoint := ApExecutionJoinPoint new
  receiver: self;
  receiverClass: Point;
  selector: #x:;
  arguments: (Array with: newX);
  sender: <vmSender>;
  sendingMethod: <vmSendingMethod>.
thisReceptionJoinPoint notNil ifTrue: [...] 
  x := newX
  thisReceptionJoinPoint notNil ifTrue: [...] 
```

where `<notSuperSend>` is some operation that searches the stack as described in Section 4.2.1, and `<vmSender>` and `<vmSendingMethod>` represent code to determine the sender and sending method, respectively, omitted here for clarity.

### 4.2.2 Advice Applicability and Dispatch at Join Point Shadows

The semantics of Apostle is that at each join point, all advice declarations are checked whether their pointcut matches the join point; if so the advice is run. To weave the program, we find shadows of the the join points, and check every advice declaration to see whether its pointcut could possibly match the shadow. For each matching shadow, we determine the residual work necessary to ensure the advice applies: some pointcuts require only statically available information to determine their relevant join points; others require dynamic runtime tests. Table 4.2 summarizes the information tested by each of Apostle’s primitive pointcuts, including which information can be statically determined.
Table 4.2: The information tested (T) by Apostle's primitive pointcuts and possibly statically known (s). Some parts of this information is able to be exposed (E) to the advice.

Since some of the properties at a join point can be statically determined, some shadows can be eliminated entirely from consideration as they cannot be statically matched by a pointcut. Other pointcuts may require run-time tests for the remaining shadows to confirm that the join point is matched by the pointcut. We call the testing that can be done statically the static testing, the run-time testing the residual testing. A residual test is simply code that tests the execution context in some manner; in fact, determining whether a join point is a method-reception can be made a simple residual test.

For example, the pointcut execution(x:) can be statically determined to match at the execution join points in methods named #x:; it cannot match the shadows of any methods named differently. The pointcut

kindOf(Set) & executions(add:) & sender(EtWindow)

cannot possibly correspond to any methods in Bag, since Bag is not a kind-of Set. But it could possibly match the method Set##add:. Truly determining whether an invocation of Set##add: corresponds a join point matched by the pointcut requires a residual test to verify that the sender was of type EtWindow.

### 4.2.3 Transforming Join Point Shadows: Dispatching Advice

Transformation of a shadow then entails in-lining the advice dispatch code for the applicable advice. This is conceptually expressed as:

```plaintext
Point>>x: newX

| thisReceptionJoinPoint thisExecutionJoinPoint |
| thisReceptionJoinPoint := ... |
| thisExecutionJoinPoint := ... |

*Dispatch applicable before advice*
```
(residual pointcut tests) ifTrue: [call before advice].
(residual pointcut tests) ifTrue: [call before advice].
(residual pointcut tests) ifTrue: [call before advice].

x := newX  "the original method body"

"Dispatch applicable before advice"
(residual pointcut tests) ifTrue: [call after advice].
(residual pointcut tests) ifTrue: [call after advice].

Since the advice at any shadow is known, the advice is simply listed one at a time.

Residual Tests

The residual tests serve to confirm that this join point is matched by a pointcut. These are Smalltalk expressions, generated by the pointcut. In the example above, the pointcut:

```
kindOf(Set) & executions(add:) & sender(EtWindow)
```

requires a residual test for the sender(). This could look like:

```
((ApSenderPrimitivePointcut
  object: ApPlatformIssues senderReceiver
  conformsTo: nil
  typeName: 'EtWindow')
  and: [(ApPlatformIssues senderReceiver class isMetaclass = false)])
```

The code ‘ApPlatformIssues senderReceiver’ is a stack-searching operation to determine the sender of the current method. This expression causes the sender to be checked that it is a kind-of EtWindow and is not a metaclass (or rather, that it is an instance, not the class).

Dispatching Advice

Dispatching advice requires determining a receiver: an aspect instance. This requires querying for the existence of the appropriate aspect instance. We finally call the advice on the aspect instance. This generally looks like:

```
((residual pointcut tests) and: [(_aspectInstance := Aspect aspectFor: self) notNil]) ifTrue: [
  _aspectInstance
  before_pcut_body: thisJoinPoint
  object: self
  exposedContext: ...].
```

There are several aspect-types, detailed in Appendix B, whose instantiation may depend on the circumstances. The call to #aspectFor: obtains the aspect instance for the receiver.

Since advice specifies a single pointcut, which may match many different pointcuts, and which exposes the same context regardless of the join point, advice receives the same number of arguments from each join point shadow. But the actual arguments vary depending on the shadow. The advice calling must retrieve and pass the appropriate values. The actual message send for the advice call is discussed next.
4.2.4 The Transformation of Advice

There are then two approaches to implementing advice, differing by where calls to advice are made. A piece of advice generally applies to a set of join points. But some subset of those same join points may also be identified by some other pieces of advice. Thus different join points may have different selections of advice, meaning there are different traversal ordering of the advice on a per-join point basis. One can choose to embed the advice traversal into the advice or into the join point shadows.

Embedding the traversal into advice requires having customized copies of the advice for each possible traversal, where each copy is customized to embed the call to the next advice for that particular traversal. This could result in a large number of copies of advice.

Apostle uses the other possibility, in which the advice traversal is embedded in the join point shadow. Advice is instead transformed into relatively normal Smalltalk methods, whose execution context is pre-computed and provided to as arguments. This allows the advice-implementations to be re-used, and are thus able to be called from a number of join point shadows.

Implementing Around Advice

Around advice is slightly different from before and after advice. Should the advice apply, then around advice can affect the remainder of the computation; but this computation must also be performed if the advice is not applicable (i.e. its residual test fails, or no aspect-instance is extant). In Apostle, this computation-remainder is encapsulated as a Smalltalk block, called a ‘proceed block.’ The around calling situation then generally looks like:

```smalltalk
unique_selector: thisJoinPoint object: _AP_receiver exposedContext1: exposedContext1 ... 
"advice body"
```

The selector name is chosen to be unique per piece of advice, so as to be able to distinguish one piece of advice from another. This is necessary as the environment and language must have some unique way to distinguish advice since advice is not named. This issues is discussed further in Section 6.1.5.

All execution context for the advice is passed as arguments to the method, including thisJoinPoint (line 1). This allows the advice-implementation to be independent from where it is called. The advice-implementation need not be customized for each shadow, and thus it is able to be called from all applicable shadows.
Lines 1–3 define the proceed-block; this is the representation of the computation-remainder. This block could itself call more advice, and define other proceed-blocks. Line 5 checks whether the around advice is applicable. If not, then the false case at line 13 ensures the computation-remainder is evaluated. Otherwise, the appropriate advice is called, provided with the appropriately translated context (as specified by the advice), and the proceed-block.

By using a block, this computation-remainder can in fact be deferred, and its return result be captured and changed — it is essentially a first-class object.

**Supporting Efficient cflow()**

Apostle’s efficient implementation of cflow() causes a marker to be set at any join points identified by its sub-pointcut. This marker is later queried by the residual tests for the advice that actually specified the cflow(). This process has been generalized such that pointcut implementations can provide customizations to the system, in the form of advice. cflow() simply defines around advice targeting its sub-pointcut, whose advice-body causes the marker to be set for the duration of the join point.

### 4.3 Incremental Weaving

Incremental programming means that any program definition can change at any time, and that ‘saving’, or effecting, the change must take little time. Supporting incremental weaving is complicated by the crosscutting nature of AOP elements. A single change to one program definition may require re-weaving a potentially large number of join point shadows. For example, consider adding the negation operator to a pointcut definition. When this change happens, all advice using this pointcut must be removed from the previously-identified join point shadows and added to the newly-identified shadows. This change requires re-weaving many shadows.

Incremental weaving (Figure 4.2) requires identifying the impact of any program change. Adding a method, hanging a pointcut, removing some advice: each change may introduce inconsistencies into the target model which must be subsequently repaired. Incremental weaving involves reconciling these changes. These inconsistencies are identified by examining the assumptions embedded into the target model.

This section proceeds by determining these embedded assumptions, and then the
possible changes in the Smalltalk environment. The remainder examines the incremental re-weaving algorithm.

4.3.1 Dependencies Amongst Language Elements

In this model, the actual source transformations occur only at methods. We have seen above that shadows have embedded calls to applicable advice and aspects, and residual tests for pointcuts too. A change to any of these elements requires regenerating the shadow. It is thus sufficient to determine what causes advice to be applicable — or no longer applicable — for a particular shadow. The following example illustrates some of the possibilities:

```smalltalk
pointcut possibleGrowth (): marshalObject () & fixedSizeMessaging()

before<possibleGrowth()>

self isFull ifTrue: [
  self error: 'full'
]
```

A change to `fixedSizeMessaging()` not only requires that advice explicitly using it be re-woven, but that any advice using a pointcut that uses it also be re-woven.

To minimize re-weaving requires identifying the assumptions compiled into the target model:

- **Shadows have embedded calls to applicable advice and aspects, and pointcut residual tests.**

- **Advice call points are identified by the pointcuts.** Advice dispatches are identified by the transitive closure of its named pointcuts. A change to any of these pointcuts requires the advice be re-woven for any previously- or newly-identified shadows.

- **Advice and pointcuts depend on the enclosing class/aspects.** A change in where a pointcut is defined, a different class or aspect, changes the scoping of any references from within the pointcut. Imagine if `fixedSizeMessaging()` was previously inherited, and subsequently overridden.

4.3.2 Types of Incremental Changes

There are several possible types of program change in the Smalltalk environment. Each has a some impact on the target model, potentially invalidating some of the weaving.

**Adding a method:** The method may be subject to advice.

**Adding advice:** All shadows at which advice applies require re-weaving.

**Adding a new pointcut**
Adding a class/aspect

Changing a method: The method may be have been subject to advice.

Changing advice: All shadows that called the advice require re-weaving.

Changing a new pointcut: Any advice specifying the pointcut must be removed from any formerly-identified shadows and added to any newly-identified shadows, which must be re-woven.

Changing a class/aspect: This is effectively the same as removing and then adding all methods, advice, and pointcuts defined by the class/aspects and its subclasses/subaspects.

Removing a method: The method may have been subject to advice.

Removing advice: All shadows dispatching the advice require re-weaving.

Removing a pointcut: Any advice specifying the pointcut must be removed from the previously-identified shadows, which must be re-woven.

Removing a class/aspect: This is effectively the same as adding/removing all methods, advice, and pointcuts defined by the class/aspects and its subclasses/subaspects.

Changing program definitions can be seen to be effectively the same as removing the old definition and adding the new definition.

4.3.3 Support for Incremental Weaving

Incremental weaving requires identifying and recording the dependencies between the various types of language element. Apostle maintains this information using three sets.

Pointcut Dependency List

This records the dependent pointcuts for every piece of advice. For each pointcut, we record its equivalent names: the different fully-qualified names referring to the same pointcut. A change to any of these pointcut by any of these names potentially requires many changes.

For example, if class $C1$ defines pointcut $p1()$, class $C2$ is a subclass of $C1$, and advice $A$ refers to $C2.p1()$, then we remember names $C2.p1()$ and $C1.p1()$ for $A$. This also includes any of the sub-pointcuts, the pointcuts depended upon by the advice-pointcut. A change to any of these pointcuts requires re-weaving $A$. 

38
Shadow List

This records the advice applicable at each shadow, meaning the advice whose pointcut statically matches the shadow. This is used when re-generating the shadow, as occurs when advice is added or removed, or the corresponding method be redefined.

Advice List

This records the advice currently in the system. This speeds determining which pieces of advice apply to newly-defined methods

4.3.4 The Re-Weaving Algorithm

Any set of changes requires they be properly ordered to update the target model according to the previously-identified dependencies. For example, removing advice using pointcut `diagramChanges()` while simultaneously redefining the pointcut requires first removing the advice dispatches from the shadows identified using the old definition of `diagramChanges()`, before instituting the new definition. To do otherwise may cause previously advised shadows to be missed. In fact, all changes are predicated on the pointcut definitions, as they identify the actual program elements to be affected — which constitutes the behaviour of the system.

The general algorithm takes as input lists of the removed and added program elements, and has three steps:

1. Process all removals: removed methods, advice, pointcuts, classes and aspects. The system will have been brought to consistency in that all shadows are identified by current pointcuts, and only make calls to existing advice.

2. Perform the pointcut redefinitions.

3. Process all additions: added methods, advice, pointcuts, classes and aspects. The system is now completely brought into consistency with respect to the changes.

For simplicity, a redefinition is treated as a removal of the old definition, and an addition of the new definition.

It is important to distinguish between named and anonymous pointcuts. Anonymous pointcuts are defined as part of advice, and are treated as either added-pointcuts or removed-pointcuts should the advice be added or removed. Named pointcuts are treated as separate elements, and must be explicitly removed.

During the course of the algorithm, modifications are scheduled for various shadows, such as adding or removing advice. These modifications are accumulated until being finally committed.
**Step 1: Process All Removals**

All removals must be processed using the old pointcut definitions so as to find and fix-up the shadows previously-identified by either removed-advice or removed-pointcuts.

1. *Delete all state recorded for shadows corresponding to removed methods.*

2. *Remove any customizations added by removed pointcuts.* This includes the pointcuts of the removed-advice too, if they are not named pointcuts; named pointcuts must be removed explicitly as they may still be used by other advice, whereas anonymous advice is not.

3. *Undo the effects of the removed pointcuts.* These include the anonymous pointcuts of removed-advice. This involves scheduling the removal of all advice using these pointcuts from the shadows identified by these pointcuts, using the old pointcut definitions.

4. *Commit all shadow modifications to this point.* These shadows must be rewoven before actually removing advice in the next step, as the action of removing advice may result in the execution of some modified shadows, inadvertently dispatching to non-extant advice.²

5. *Remove the removed-advice.* Remove the actual advice implementation methods.

6. *Delete the pointcut dependencies for any advice using a removed-pointcut.* The actual advice affected should be remembered, if they are not being removed; they may still be applicable due to added-pointcuts. These dependencies are rebuilt in Step 3.1.

At this point, all calls to removed-advice, or advice identified by removed-pointcuts, have been removed with the regeneration of the relevant shadows in Step 1.4. The target model is now internally consistent.

**Step 2: Perform Pointcut Redefinitions**

The new definitions are made current. Pointcut deletions are treated as equivalent to re-defining the pointcut to nothing.

**Step 3: Process All Additions**

This essentially mirrors Step 1, except that additions use the new pointcut definitions.

²This occurred during the Apostle’s development with disastrous consequences as the compiler was affected, thus preventing any recovery.
1. *Update pointcut dependencies for new and re-woven advice.* This includes the advice whose pointcuts were removed/redefined in the previous phase.

2. *Add customizations for new pointcuts.* This includes the anonymous pointcuts of new-advice.

3. *Install the new advice.* Install the advice implementation methods. These must be in-place before re-weaving the modified shadows in Step 3.6.

4. *Use the new pointcut definitions to identify possibly applicable shadows.* Schedule the applicable advice for the identified shadows. The pointcuts of new advice are considered as new pointcut definitions.

5. *Process the new or updated methods.* Walk all known pointcuts, scheduling the applicable advice for any matched shadows of the new/updated methods.

6. *Commit the shadow modifications.*

### 4.3.5 Analysis of the Re-Weaving Algorithm

This section demonstrates that this algorithm correctly processes changes and updates the target model and associated information sets. Referring to the changes identified in Section 4.3.2, we first show that the algorithm correctly processes any single change. Having satisfied this, it then shows that any two simultaneous changes are also correctly processed.

**Re-Weaving Single Program Changes**

*Adding or removing a pointcut is effectively the same as changing its definition from or to a ‘nil’ pointcut, which matches no pointcuts.*

**Adding a new method:** Step 3.5 ensures that new methods are checked against advice for applicability; step 3.6 then commits any modifications for such identified methods.

**Removing a method:** Step 1.1 ensures that all state is forgotten for removed methods.

**Replacing a method:** Step 3.5 causes updated methods to be processed as if they were new. Thus all advice is checked for applicability.

**Adding advice:** Step 3.1 ensures that any pointcut-dependencies are recorded. Step 3.2 adds whatever customizations are required of the advice’s pointcut. Step 3.3 causes the new advice to be installed; it is now safe to dispatch to this advice. Step 3.4 identifies any possibly relevant shadows, and Step 3.5 ensures that any new methods are also checked. Step 3.6 finally commits all modifications for the shadows.
Removing advice: Step 1.2 removes whatever customizations are required of the advice’s pointcut. Step 1.3 removes the advice from all previously-identified shadows; Step 1.4 actually rewrites these shadows; there are no possible dispatches to the advice. Step 1.5 causes the advice-method to be removed. Step 1.6 ensures that any pointcut-dependencies are forgotten.

Changing a pointcut: Step 1.2 removes whatever customizations were required of the pointcut. Step 1.3 removes the advice using the pointcut from all previously-identified shadows, while Step 1.4 rewrites these shadows; there are no traces of advice dispatches; the new pointcut definition will be used to identify the new shadows. Step 1.6 ensures that any pointcut-dependencies are forgotten. Step 2 performs the pointcut redefinition: the new definition is now official. Step 3.1 rebuilds any pointcut-dependencies for advice using this pointcut. Step 3.2 adds whatever customizations are required of the new pointcut. Step 3.4 identifies any possibly relevant shadows, and Step 3.5 ensures that any new methods are also checked. Step 3.6 finally rewrites these shadows with advice-dispatch calls.

Re-Weaving Multiple Simultaneous Changes: Changing Class and Aspect Definitions

Changing class or aspect definitions are more complex, but can be reduced to considering them as a set of simultaneous operations. Adding a class or aspect is equivalent to adding all of its defined elements, methods, pointcuts and advice. Removing a class or aspect is similarly equivalent to removing all elements. Changing a class can be thought of as first removing all elements, and then re-introducing them. In fact, changing any element can be thought of in this way: first removing the old definition, and then adding the new one.

Steps 1 and 3 are essentially stand-alone, each bringing the target model back to a state of internal consistency. Thus any removal and addition change is correctly ordered. This section lists the possible combinations, and shows that any two simultaneous removals or two simultaneous additions are processed correctly. Then the ordering of changes does not matter: they are idempotent.

Adding pointcut and method: It is essential that the new pointcut definition be in place before considering the new method to ensure the method would be identified if applicable. This is ensured by having the new pointcut definitions take effect in Step 2, before methods are considered in Step 3.5.

Adding pointcut and advice: It is similarly important that new pointcut definitions be in place before their use by advice — otherwise the advice may wrongly target certain join points. This is similarly ensured by the pointcut definitions being effected in Step 2, before advice is considered in Step 3.4.
**Adding method and advice:** New methods are checked against all advice for advising in Step 3.5, after any new advice has been installed (Step 3.3 and 3.4). They will thus be successfully advised.

**Adding two advice:** The only possible complication arises when the advice target some commons set of join points. Steps 3.4 and 3.5 are responsible for associating advice to join point shadows, and step 3.6 re-weaves the shadow.

**Adding two pointcuts:** The only possible complication arises when one pointcut is dependent on the other, as identifying the shadows requires both definitions to be up-to-date. This is ensured since the pointcut definitions are made official in Step 2, but not used until later.

**Removing pointcut and method:** The order of these two operations does not matter so much, as the method is removed regardless. But Step 1.1, which restores the old shadow, occurs before pointcuts are removed (meaning redefined to nothing) in Step 2.

**Removing method and advice:** Like simultaneous removal of a pointcut and method, the ordering of these two removals aren’t terribly important. The method is removed regardless.

**Removing pointcut and advice:** The old definition of the pointcut must be used to identify the shadows where the advice was applied. This is ensured by having advice removed in Steps 1.3, 1.4 and 1.5, before the pointcut removal in Step 2.

**Removing two advice:** The only possible complication arises when the advice target some commons set of join points. Steps 3.4 and 3.5 are responsible for associating advice to join point shadows, and step 3.6 re-weaves the shadow.

**Removing two pointcuts:** The only possible complication arises when one pointcut is dependent on the other, as identifying the shadows requires the old definitions. This is ensured as the shadow-removals of Step 1.3 are done before the new pointcut definitions are only made official in Step 2.

Thus by showing that any single change is properly effected, and that any two changes can be properly effected, we can reason by induction that any number of changes can be properly effected.

### 4.4 Summary

This chapter has presented some of the more interesting aspects of the implementation strategy of Apostle. It has described how advice is transformed and transforms the image.
It presented the algorithm used to apply changes to the image incrementally.
Chapter 5

Results and Evaluation

This chapter presents an evaluation of the Apostle language and implementation. Section 5.1 presents some results of having restructured three small Smalltalk applications. It finally presents a brief qualitative evaluation in Section 5.2 against the criteria identified in Sections 2.1.6 and 3.1, including attempting to gauge the speed impact when programming with Apostle.

5.1 Three Case Studies: Using Apostle For Application Restructuring

The following section describes using Apostle for restructuring three small applications.

5.1.1 Aspectizing a Model/View/Controller Application

The first body of code involves a simulation demonstrating the existence of critical phenomena in asynchronous systems [19]. Several crosscutting concerns were isolated from this application. Many of the design decisions taken in Apostle were motivated while trying to restructure this application. Much of the resulting aspectized code has been used as examples in this thesis for demonstrating various language features.

The simulation is loosely architected as a model-view-controller system [31]. A simulator tracks a set of nodes, which move between three states: waiting, scheduled, and fired. The progress of the simulation, such as nodes being scheduled for execution, is reflected by a GUI. The GUI also serves as a controller, responsible for configuring, starting and stopping the simulation.

Simulators are represented as subclasses of PercolationSimulator. This base class implements most of the simulation infrastructure, maintaining information on nodes and informing the observers of changes. The subclasses represent different network topologies,
such as rings and meshes, and deal with the details of scheduling nodes: a node can only be scheduled depending on the state of its neighbours, and neighbours depend on the network.

*SimulationWindow* implements the simulator view. It supports two drawing operations. `#redraw` causes the entire view to be redrawn. `#drawNode:` updates a single node, potentially requiring the entire view to be redrawn.

**Ridding the Application of Subject/Observer**

This separation of model and view enables the the simulator to be run stand-alone. But the view then requires some mechanism to be alerted to any changes, such as through a publish/subscribe mechanism like Subject/Observer pattern [15]. But this requires littering the simulator with seemingly-extraneous calls to alert its observers such as:

```smalltalk
fire: event
    event node fired.
    observers do: [:observer | observer eventFired: event]
```

There are two difficulties with this form of subject/observer. First, all observers must observe all event types or else typing errors occur (manifesting themselves as `#doesNotUnderstand:` errors in Smalltalk). This could be remediated with a proper event system. More importantly, observers can only latch onto previously identified events. To observe new events requires modifying the source code to insert similar observer-alerting code in appropriate places; this may not be desirable — or possible.

Both problems are easily tackled with Apostle. Rather than maintain the subject/observer infrastructure, the simulators publish individual pointcuts for their different conditions:

```smalltalk
pointcut eventFired(simulator, event):
    kindOf(PercolationSimulator simulator) & receptions(#fire: event)

pointcut eventScheduled(simulator, event):
    kindOf(PercolationSimulator simulator) & receptions(#scheduled: event)
```

Interested parties are now able latch onto only those events of interest by specifying some particular pointcut. New events are easily added as new pointcuts.

**Completing the Observers.** But *SimulationWindows* monitor only one particular simulator; the pointcuts identify all simulators. The simplest solution continued using the subject/observer style, following the observer aspect as proposed by Kiczales [26] where the pointcuts identify events.

A base aspect, *ApObservingAspect*, provides some plumbing for implementing subject/observer as an aspect. Observing aspects are singleton aspects that maintain a mapping between subjects and their observers.

```smalltalk
ApAspect subaspect: #ApObservingAspect
```

46
classVariableNames: ''

poolDictionaries: ''

addObserver: observer of: subject

^((observers at: subject ifAbsentPut: [ApPlatformIssues newWeakSet]) add: observer

observersOf: subject

^observers at: subject ifAbsent: [#()]

removeObserver: observer of: subject

^((observers at: subject ifAbsent: [^nil]) remove: observer ifAbsent: [nil])!

initialize


^super initialize

The association of observers to subjects is identified by the associations() pointcut, declared as nil. This identifies it as requiring to be over-ridden in any sub-aspects.

pointcut associations(subject, observer): nil

This aspect does not provide a sample implementation of the associating advice: the possible policies for determining what should occur with previous subject/observer relationships is complex, and best addressed by the application’s implementation (as seen below).

The SimObserving aspect then adds the necessary definition for associations(), and tags advice to the various observable conditions published by the simulators.

ApObservingAspect subaspect: #SimObserving

instanceVariableNames: ''

classVariableNames: ''

poolDictionaries: ''

pointcut associations(subject, observer):

kindOf(SimulationWindow observer) & receptions(simulator: subject)

around<associations(simulator, window)>

"Associate the window as an observer of the simulator, first forgetting any previous observing associations held by the window."

window simulator notNil ifTrue: [

aspect removeObserver: window of: window simulator.

^[PROCEED] ensure: [aspect addObserver: window of: simulator]

after<PercolationSimulator.eventFired(simulator, event)>

(aspect observersOf: simulator) do: [:observer | observer eventFired: event]

after<PercolationSimulator.eventScheduled(simulator, event)>

47
As a result, all simulators are clean of any subject code while the equivalent functionality is still present. Adding new events to track is a matter of identifying new pointcuts and some simple advice — the simulator code need not be touched, except for perhaps some better refactoring. This still requires the observers to understand all events.

Optimizing Window Redrawing

SimulationWindows are notified as significant node-events occur, and update their view accordingly. The SimulationWindows attempt to be intelligent, drawing only the nodes that have changed. But sometimes the window must be redrawn, as its current view has become out-of-date (i.e. the simulation has scrolled off the screen). Events may still occur during the redraw. The following aspect introduces an optimization to avoid pointlessly queueing further individual node-updates when the entire window is redrawn.

First we define the aspect:

```smalltalk
Ap Aspect subaspect: #SimDrawingOptimization
perObject: 'kindOf( SimulationWindow )'
instanceVariableNames: 'redrawing'
classVariableNames: ''
poolDictionaries: ''

Advice is defined to track when a redraw is in progress.
around<kindOf( SimulationWindow ) & receptions(#redrawGraph)>

redrawing := true ifTrue: [^self].
redrawing := true.
^[PROCEED] ensure: [redrawing := false]
```

There's no point performing a redraw when one is in progress.

Individual draw-nodes should be discarded when a redraw is in effect.
around<kindOf( SimulationWindow ) & receptions(#drawNode:)> 

```smalltalk
redrawing := true ifTrue: [^self].
^[PROCEED]
```

Serializing GUI Updates

VA/ST requires that all graphics operations be serialized by having all such operations be executed from a particular process, called the UIProcess. But as simulators are active [48], meaning they have a separate process driving the simulation, the graphic updates will not be performed through the UIProcess. Fortunately VA/ST provides a helper method CwAppContext3#asyncExecInUI: that schedules a provided block for execution by the UIProcess.
Thus one possible solution would then be to wrap all graphics operations with calls to \texttt{CwAppContext\#asyncExecInUI}. For example, \texttt{#redraw} could be implemented as:

\begin{verbatim}
SimulationWindow\#redraw

CwAppContext default execlnUI : [self redrawGraph].
\end{verbatim}

But the presence of the calls to \texttt{#execlnUI} is orthogonal to the purposes of the method: it is crosscutting.

Rather than litter the window code with calls to this method, all UIProcess-serializing knowledge is captured by the following piece of advice:

\begin{verbatim}
around<kindOf (SimulationWindow) & receptions (#redraw)>

CwAppContext default execlnUI : [PROCEED].
\end{verbatim}

\texttt{#redraw} is then simply:

\begin{verbatim}
SimulationWindow\#redraw

self redrawGraph
\end{verbatim}

since there were multiple methods initiating such requests, these methods were specified in a separately named pointcut \texttt{SimulationWindow:uiRequest()},

\begin{verbatim}
pointcut uiRequests () :
kindOf (SimulationWindow) &
(receptions (#eventFired :) | receptions (#eventScheduled :) |
receptions (#redraw) | receptions (#updateInformationDisplay))
\end{verbatim}

The advice was modified to use this new pointcut:

\begin{verbatim}
around<SimulationWindow.uiRequests()>

CwAppContext default asyncExecInUI : [PROCEED]
\end{verbatim}

5.1.2 Progress Monitoring

A course project required generating the entire state space for a particular problem using a dynamic programming solution. As this generation could take several hours, the generator was refactored to report its progress through a progress meter. Although eventually isolated to requiring changes in only three methods, the calls were intrusive.

The first change modified the method initiating the generation to first create the progress meter. The remaining changes were to the methods implementing the two phases of the program, whose program loops were modified to issue periodic calls to \texttt{#fractionComplete} and \texttt{#fractionComplete:messageString} to advance the progress meter. For example:

\begin{verbatim}
generateTransitions

| descriptions considered |
| transitions notNil ifTrue : [^self].
\end{verbatim}
transitions := LookupTable new.
considered := 0.

"Why do this in reverse? Because we secretly know that the last has the
largest number of s's at the end. So this way we get the keys having
maximum number of s's."

self
fractionComplete : 0
messageString : 'Generating all possible transitions...'.
(descriptions := self generateAllPossibleDescriptions) reverseDo:
[:description | | pc pew |
  self fractionComplete : (considered := considered + 1) /
  descriptions size.
  pc := (ProcessorConfiguration for: description) canonicalize.
  pew := pc equivalenceWrapper.
  transitions at: pew ifAbsentPut : [ pc calculateTransitionProbabilities]

Although this was subsequently refactored to be made much cleaner, the calls to #fraction-
Complete: were still extraneous to the purpose of the generator.

This was isolated to the following aspect. Since the meters were particular to each
generator, the aspect is a per-object aspect:

ApAspect subaspect: # StatusUpdating
perObject: 'kindOf (GenerationalGraph)'
instanceVariableNames: 'graph progressDialog '
classVariableNames: ''
poolDictionaries : ''

When the generation is initiated, open a progress dialog. Close it once finished.

around<kindOf (GenerationalGraph) & receptions (#generate)>

  graph := self.
  aspect openProgressDialog.
  ![PROCEED] ensure: [aspect closeProgressDialog]

Reset the meter to zero when beginning either of the two phases of the generation:

before<kindOf (GenerationalGraph) & receptions (generateTransitions)>

  aspect
  fractionComplete : 0
  messageString: 'Generating all possible transitions ...

before<kindOf (GenerationalGraph) & receptions (generateSolution)>

  aspect fractionComplete : 0 messageString: 'Generating solution matrices ...

The generator underwent a minor refactoring to push the respective calculation steps
into new methods, suitable for observing. At each step, update the progress meter:

after<kindOf (GenerationalGraph) & receptions (considerTransitionsFromDescription : number: considered of: total)>

  aspect fractionComplete : considered / total

after<kindOf (GenerationalGraph) &
receptions (#calculateSolution: index: currentIndex transitionProbabilities:)

aspect fractionComplete: currentIndex / graph indexMapping size

The generation code is clean of any progress-tracking code. The minor refactorings to better support the aspect are also considered as improvements to the code.

5.1.3 Adding Unanticipated Functionality to a Closed Framework

Smalltalk has always been a rather mouse-centred programming environment. This causes problems for the author, who suffers from repetitive strain injuries (RSI) due to bad mousing practices. Since the environment is itself written in Smalltalk, an ideal enhancement would incorporate more keyboard shortcuts into the browsers. This example used Apostle to better incorporate these changes.

After some investigation the simplest solution appeared to be the use of keyboard accelerators. These are unique keyboard combinations associated with menu items, whose use invoke the menu item. For example, Control-S may be bound to File → Save, or Control-F to Edit → Search. As menus and accelerators must be built at creation time, it was necessary to find and modify the relevant menu-creation sites in the browser framework to add appropriate accelerators.

Extending the Framework through Unanticipated Overriding

The ENVY/Smalltalk browser sub-framework is implemented as a very deep class hierarchy inheriting from EtWindow, with every type of browser represented as a class. Examination showed that a browser’s menu bar is created when sent #createMenusAfterWorkRegion, which was overridden appropriately by the various browsers. Unfortunately the implementations of #createMenusAfterWorkRegion provided no support for adding new menus. In fact, the VA/ST browser set did not seem designed to ever be extensible.

While these implementations could be modified to perform the appropriate menu-creation, this required changing shipped code. But this would preclude sharing the solution with others, as few would consider modifying shipped code, especially code as fundamental as the browsers. The ideal solution would be packaged as a set of class extensions in an ENVY/Manager Application requiring no method-modification, such that anybody could load them.

The Non-AOP Solution

Fortunately there were few implementations of #createMenusAfterWorkRegion. Thus the required changes could be accomplished by ‘inserting’ identically-named methods in appropriate places in the hierarchy such that all other browsers inherited this automatically.
Figure 5.1: The ellipsis acts as a visual cue that other configurations of this button have information.

The following method was added to each of the immediate subclasses of *EtWindow*:

```plaintext
createMenusAfterWorkRegion

| menu |
super createMenusAfterWorkRegion.
self bdaAddOptions : (menu := self newMenu).
menu title : 'Hacks'.
sel menuBar addMenu: menu.
```

#*bdaAddOptions*, so named so as to not possibly clash with any existing methods, were added to appropriate classes in the hierarchy, responsible for adding the various short-cuts appropriate for their respective classes.

Framed as a general blueprint, this solution found places in the hierarchy where we could intercept important methods to tack on our own code, before continuing on to the superclass’ implementations. This could be thought of as “poor-man’s advice.” It was used for several other additions, notably for adding visual cues to various user interface items (e.g. Figure 5.1).

**The AOP Solution**

The method-overriding is easily replaced with advice:

```plaintext
after type (EtWindow) & receptions (#createMenusAfterWorkRegion)>

| menu |
self bdaAddOptions: (menu := self newMenu).
menu title: 'Hacks'.
self menuBar addMenu: menu.
```

The visual cueing was similarly reworked.

While not a dramatic restructuring improvement of the OO solution, it is still an improvement. The code now depends on the protocols followed within the sub-framework rather than be dependent on the implementation status of the sub-framework.
5.2 Subjective Evaluation

We also try to evaluate Apostle as to whether it preserves the required properties of Smalltalk environment, as identified in Section 2.1.

**Immediacy/Responsiveness**  Apostle preserves the immediacy property. Saving an aspect definition causes the change to be effected immediately. Saving a pointcut causes immediate application, or realignment of the image with its definition. New advice is put into effect. No further compilation steps are required.

**Incremental** The healing algorithm presented in Section 4.3.4 processes changes idempotently. Thus the algorithm is truly incremental.

The ordering in defining methods and pointcuts identifying those methods does not matter: they are idempotent. Different orderings have the same net effect: the appropriate methods are properly targeted with the applicable advice. This preserves the incremental property.

**Reflective** Apostle features both declarative and reflective facets. Pointcuts, advice, and aspects are all be first-class objects, and manipulable within the environment.

**Imperceptible** Apostle’s performance characteristics are considered secondary to correctness. But it seems adequate in use for day-to-day tasks. Table 5.1 shows some very preliminary timings for two operations: saving a method, and applying simple advice that targets only a single method; these were obtained using Apostle on a 700MHz Pentium-III running VAST 5.5.2 for Linux. These were chosen not only because they are easy to time, but also because they are relatively indicative of the majority of program changes while programming in Smalltalk. The advice application is surprisingly less than the time required by the environment to perform the save. One possibility is that the Apostle code causes a different form of the method compilation code than would the normal ENVY/Manager save operation.

Qualitatively, saving methods is generally quick, though adding instance methods on any hierarchy of Behavior can cause a pause, as these require processing all the classes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saving one method</td>
<td>42.3ms</td>
</tr>
<tr>
<td>Saving one method with 20 advice</td>
<td>37.8ms (!)</td>
</tr>
</tbody>
</table>

Table 5.1: Usability Timings
in the image (upwards of 250). Advice save-time is proportional to the number of elements affected, but even the \texttt{#instVarAt}: examples of Section 3.2.7, whose pointcut requires walking the entire image, seems reasonably quick.
Chapter 6

Conclusions and Future Work

This thesis has presented the design and implementation of Apostle, an aspect-oriented language extension for Smalltalk. It presented the language and described how its presentation fits with the environment. It also described an implementation that preserves the requirements of the Smalltalk environment, especially its incremental nature. These requirements were summarized as:

**Immediate:** The language bindings must preserve the immediacy, or liveness, expected of the environment: changes are immediately effected.

**Incremental:** The environment must continue to promote incremental development, that changes can occur in any order and at any time.

**Reflective:** The tools must support reflective facilities to facilitate building and enhancing tools.

**Imperceptible:** The programmer should not be aware of any separate compilation process.

There were three key requirements to cause the implementation to succeed:

1. the dependencies and assumptions made in the model must be well understood;
2. the possible changes to the program must be analyzed to understand how they break the dependencies and assumptions;
3. the implementation must somehow obtain notification of all changes to the program, so as to correct the model.

### 6.1 Future Work

The work on Apostle has spawned a few questions.
6.1.1 Lazy Installation

The strategy outlined here follows the traditional compilation route where each change, such as redefining a pointcut or adding new advice, is done in full. Considering that some changes have large footprints, compilation is slow,\(^1\) and Smalltalk's incremental nature may lead to a method to be changed many times before ever executed, then acting in a lazier could prove to be a big win. This is somewhat similar to Just-in-Time compilation: further processing is delayed until execution demonstrates its need. One possibility might be to identify and mark the locations affected for processing once execution actually reaches that point by replacing affected CompiledMethods with some other variant that triggers its recompile.

Similar trade-offs exist for this solution as with JIT technologies. These solutions require additional complexity, and possibly forgoing optimizations available to the system. As this JITting would be done within the image, this would raise further possible interactions between Apostle and its environment. Consider the effects should some method essential for Apostle's or the system's functioning be so identified and marked.

6.1.2 Other Incremental Environments

This work may perhaps be more broadly applicable than simply incorporating aspect-oriented technologies in programming environments, but to any incremental environment. These are environments where a program can be dynamically extended after compile-time, such as replacing portions of the program, or by dynamically loading new components. Such environments require additional support for AOP, as these program portions may be targeted by the aspect programs. Indeed, portions of the program may also be removed, including AOP constructs, possibly requiring removal of aspect programs. Drawing general conclusions from this implementation is difficult: consider prototype-based languages such as self [49].

6.1.3 Side-Effects Arising From Incremental Changes

Incremental development with AOP can introduce previously-unseen changes to program semantics. Consider defining a cflow() pointcut identifying a situation — that is the midst of occurring before the definition is effected. Presumably advice targeting this cflow() should begin being executed.

This is perhaps better illustrated with an example. Consider the following piece of advice.

\(^1\)Brant et al. claim that the compilation route of wrapping methods (i.e. generating the method wrapper as compiled source; the method used by Apostle) is by far the slowest method due to the compilation overhead [7].
This advice is to be executed while in the context of any join points identified by \( p1() \).

Now consider redefining \( p1() \) to:

\[
\text{pointcut } p1(): \text{kindOf}(\text{Process})
\]

Since the environment itself runs in a process, then this advice becomes applicable after the redefinition, and the advice body should be executed. The current Apostle implementation does not provide this.

Identifying these implementation-driven differences in semantics is important.

### 6.1.4 Use and Misuse of Types in Identifying Join Points

There is some concern in the provision of the \( \text{kindOf()} \) primitive designator. Smalltalk convention recommends against the use of \( \#\text{isKindOf:} \) in code to determine affect behaviour: it confuses behaviour with implementation. Programmers are instead encouraged to query an object whether it supports some protocol, such as asking \( \#\text{isPointcut} \).

Providing support for this in the pointcut language causes efficiency problems. Consider the following pointcut:

\[
\text{supports}(\#\text{isPointcut}) \& \text{receptions}(\#\text{at:})
\]

This pointcut must necessarily match all the classes in the image, as all classes support the message \( \#\text{at:} \). Protocol-testing messages, such as \( \#\text{isPointcut} \), are generally sent to object instances at run-time. Pointcuts, however, are evaluated statically at compile-time, when no such object instances exist. This new pointcut \( \text{supports()} \) is a purely dynamic pointcut, requiring dynamic tests to be placed throughout the pelements identified by the remainder of the pointcut. Thus protocol-testing will cause dramatic increases in the implementation foot-print (meaning the size of the image affected), thus increasing the running time — both at compile-time and run-time.

This problem arises as Smalltalk has no explicit notion of interfaces. There have been some attempts to provide such abilities, such as SmallInterfaces [42]. Type-inferencing is also a potential solution.

### 6.1.5 Naming Advice

Apostle advice does not fit well with the Smalltalk environment. The browsers rely on having some tag to identify appropriate program definitions, such as class or method names. The browsers do not actually know that a replacement is about to occur upon a save: the old definition happens to be overwritten upon successful compilation.
This ultimately prevents replacing, or updating, advice. Since Apostle, following AspectJ, does not name advice, there is no way to uniquely identify a particular piece of advice within an aspect. An aspect may have several pieces of advice using either the same pointcut, or the same advice-body, or same advice-types. Since advice has no unique tag, advice can never be perceived as having been changed: the system cannot know whether to replace existing advice or add as more advice to a pointcut. Replacing advice requires removing the original definition and adding the new definition.

Advice could be named. But this then poses questions on the semantics of inheriting and overriding advice. What does it mean if a sub-aspect overrides its super-aspect's advice? Should the overridden advice not be executed when applicable? Or should all applicable advice be executed? How can a subaspect customize its superaspect's advice, if the super's advice is has fully-specified elements (i.e. pointcut not resolved by scoping). Should inherited advice be effected from its inherited scope, effectively new advice with a potentially different pointcut?

This is different from object-oriented programming’s class inheritance and method overriding as they only have effect when the class is actually instantiated and used. This instantiation is under program control: if a the overriding subclass is never instantiated, then the overridden methods are never used. Advice, as defined by Apostle and AspectJ, are put into effect once an aspect is instantiated, which can occur at any time.
Bibliography


Appendix A

A Simple Smalltalk Tutorial

This appendix aims to impart a brief understanding of Smalltalk-80 [17], considered the standard form of Smalltalk.

A.1 Fundamentals

In Smalltalk, *everything* is an object, including classes and methods. It is a dynamically typed language: the types of arguments are unspecified, being determined at run-time rather than compile-time. This dynamism is possible as Smalltalk is interpreted by a virtual machine (VM), essentially an idealized CPU abstraction implemented in software.

Smalltalk is a class-based object system, where objects are instances of classes, as compared to prototype-based systems such as self [49], in which objects are cloned from other, prototypical instances.

There are five kinds of literals in Smalltalk: booleans, numbers, individual characters, symbols, and arrays of literals. Literals have the property that, regardless of where they occur, they always represent the same object. 'true' and 'false' are boolean literals. '1' and '2' are numeric literals. Character literals are indicated by the '$' prefix; '$a' is the literal for 'a'. Symbols are a unique strings, indicated by an octothorpe prefix as in #printString; they are used for various purposes by the virtual machine, such as for method names.

There are four fundamental language constructs to the Smalltalk language: variables and their assignment, message sends, returns, and blocks. These are discussed in turn.

A.2 Variables and Their Assignment

Variables must be declared before they are used, and are done at the top of the current scope. The following declares a variable named 'x':

```smalltalk
|x|
```
Variables are assigned using `:=`, as in:

```plaintext
|x|  
x := 'a string'
```

All assignments in Smalltalk are by reference.

## A.3 Sending Messages and Return Values

Smalltalk describes execution in terms of sending messages to an object, or equivalently making requests of an object. A message is composed of a selector, used to determine the method, and the arguments to the method. The object to whom a message is sent is referred to as the receiver. Selectors are represented by symbols; note that the selector is only a name, different from the method itself.

### A.3.1 A Simple Example

Consider the following code-snippet, or `doit`:

```plaintext
1 + 2
```

When evaluated, this causes the selector `#+` to be sent to the object `1`, a literal instance of `SmallInteger`, with an argument of `2`. `SmallInteger` maps `#+` to the appropriate method, which in this case may direct that the message be handled internally by the VM. This example evaluates, not too surprisingly, to `3`.

This expression forms a statement. Statements are generally ended with a period `.`. The period can be omitted from the last statement, as shown above.

### A.3.2 A More Complex Method

Now consider the following more complex example, the definition of a method named `#result`:

```plaintext
result

| stream |
stream := self debuggingStream.
stream next: 4 put: $=.
stream nextPutAll: '> '.
stream nextPutAll: (1 + 2) printString.
^stream
```

The first line is the method name, for a unary method named `#result`. There are three types of methods: unary, binary, and keyword. Unary methods have an alphanumeric name and whose only 'argument' is the receiver itself. Binary methods have non-alphabetic names and generally represent binary operators like `+' that take an argument in addition to the
receiver of the message. Keyword methods take any number of arguments; we see an example shortly.

**Unary messages** Line 5 features the first message send. `self` is a special keyword bound to the current receiver, the object to whom `#result` will have been sent. This line assigns to `stream` the result of having sent `#debuggingStream` to the receiver. Note that each statement is terminated by a period.

**Keyword messages** Line 6 features the first keyword message. A **keyword selector** is an ordered list of alpha-numeric, colon-terminated keywords with as many arguments as keywords. Each argument appears after their associated keyword. So the keyword message, `#next:put`, is a two-argument keyword, whose first argument is the integer `4`, and second is the character literal for `='`. This message means 'put four consecutive copies of `=` on the receiving stream.'

The next line, line 7, also features a keyword message. `#nextPutAll:` puts all the elements of the provided collection on the receiver, in this case a literal string. Strings are a collection of characters, and begin and end with a single quote.

**Order of precedence of messages** Line 8 features all three message types, and illustrates message precedence. Messages of the same type are evaluated in left-to-right order, subject to parentheses. Different message types are evaluated using the following order of precedence: unary, binary and then keyword, subject to parentheses. This is why

```
1 + 2 * 3
```

results in 9, not 7.

Line 8 then first causes the sum to be performed due to parentheses. `#printString` is then sent to the sum result, and this result is then provided as the sole argument to `#nextPutAll:`.

**Returning values** The `^` of line 9 causes the method to return with the value of the provided expression. Methods can only return a single object; the receiver is returned in the absence of an explicit return.

**Cascaded messages** The messages of lines 6–8 are all sent to a common receiver. These can be rewritten as a set of cascaded messages, as follows:

```
self debuggingStream
   cr;
   next: 4 put: $=;
   nextPutAll: '> ';
   nextPutAll: sum printString.
```
Cascaded messages are indicated by terminating the previous message with a semi-colon (';'), which means that the next message should be sent to the same receiver. This is more typing-friendly, and allows the compiler to perform some optimizations.

Use of protocols  The meaning of these messages sent to the stream is clear as they are well-known messages of the \textit{WriteSteam} protocol. Protocols are discussed further in Section 2.1.

A.4  Blocks

A block is a set of program statements, surrounded by square brackets, that carries its definition context. They are a manner of representing deferred execution, and are \textit{full-closures}. As demonstration, we consider how Smalltalk handles conditional constructs.

Smalltalk is such a simple language that it has no separate if-then or if-then-else constructs. Rather, it has two classes, \textit{True} and \textit{False}, both subclasses of \textit{Boolean}, which understand the common selectors \#ifTrue:ifFalse:, \#ifTrue:, and \#ifFalse:. These take blocks as arguments, representing the actions to be done given the appropriate outcome. For example:

\begin{verbatim}
string := string isUppercase ifTrue: [string] ifFalse: [string asUppercase].
\end{verbatim}

The implementations of these methods are simple, invoking their corresponding block. Implementations on \textit{True} only fire the provided \textit{trueBlocks}:

\begin{verbatim}
True>>#ifTrue: trueBlock ifFalse: falseBlock
  ^trueBlock value

True>>#ifTrue: trueBlock
  ^trueBlock value

True>>#ifFalse: falseBlock
  ^nil
\end{verbatim}

while the implementations on \textit{False} only fire the \textit{falseBlock}:

\begin{verbatim}
False>>#ifTrue: trueBlock ifFalse: falseBlock
  ^falseBlock value

False>>#ifTrue: trueBlock
  ^nil

False>>#ifFalse: falseBlock
  ^falseBlock value
\end{verbatim}

66
We can see the deferred execution represented by the block is only performed if sent the \#value message. The block has the context of its definition point; thus self in the block '\[self\]' will be bound to the receiver where the block was defined, and not where it is evaluated.

Blocks can also be provided arguments, but this is not addressed here.

### A.4.1 Out-of-Context Returns

Blocks can also contain return statements ('\A'). These cause a return from the method that defined the block. Consider the following methods, implementing a thesaurus:

```smalltalk
includes: word
  "Answer true if a definition for \@word is present, false if none."
  self lookup: word ifAbsent: [^false].
  ^true

lookup: key ifAbsent: absentBlock
^dictionary at: key ifAbsent: absentBlock
```

Should the method `Dictionary\#at:ifAbsent:` (called in line 9) evaluate its second argument as a result of being called from \#lookup: (line 4), then \#includes: will immediately return with false. This effect will occur whenever that block is evaluated, regardless of how deep execution has proceeded during the execution of \#lookup:ifAbsent:.

This behaviour is exploited throughout the Smalltalk class library, especially with the \#ifTrue:ifFalse: control structures. Consider the following method:

```smalltalk
log: string

logger isNil ifTrue: [^self].
logger addEvent: (LoggingEvent text: string)
...
```

### A.4.2 Special Block Messages

Blocks support two other important messages: \#ensure: and \#ifCurtailed:, both of which take another blocks as their argument. \#ensure: guarantees that its second argument will execute after evaluating the receiver; it serves the same purpose as Java's try-finally construct. \#ifCurtailed: executes its second argument if the receiver does not complete normally, either from an out-of-context return or an exception.

Apostle uses \#ensure: to ensure after advice is called should execution reach the actual join point.
A.5 Classes

Classes define the shape of their instances (the number of instance variables, whether it is variable, etc.), in addition to their behaviour (through the methods available). They are created by sending a creation message to their proposed superclass. For example:

```
Collection subclass: #Bag
    instanceVariableNames: 'occurrenceDictionary'
    classVariableNames: ''
    poolDictionaries: ''
```

This creates a class named Bag. Only single inheritance is supported, and almost all classes inherit from Object.

Instance variables, or inst-vars are variables that are independent between instances; they are the per-instance state. Class variables are common across the class. Convention has inst-vars beginning with a small letter and class-vars with capital letters. Class-vars are common to all subclasses too; a separate variant, class-inst-vars, are allocated on a per-class basis.

A.5.1 Creating Instances of a Class

An instance of a class is generally created by sending #new to the class. The class may override this message to perform per-instance initialization, as in:

```
new

^super new initialize
```

This causes #initialize to be sent to the new instance before being returned to the creator. Of course the class must implement #initialize.

A.5.2 Reflection and the Metaclass Jungle

Smalltalk is a reflective language [41]. Not only can classes and methods can be looked up dynamically, but objects can be easily introspected. It is easy to determine the number of slots allocated to a particular object.

Smalltalk, as noted earlier, is a class-based system. Each object has an associated class. Interestingly, this class can be changed, though this is not recommended for the weak at heart! This technique is used by several of the current aspect solutions described in chapter 2.

Classes are themselves instances of other classes, specifically customized instances of Metaclass. The importance is that all classes inherit from Behavior, which defines many of the interesting reflective protocols. We will not follow this train of though any further: it is very confusing and adds nothing to this discussion.
A.6 Other Miscellany

Transcript is a global object serving an equivalent purpose as C’s stdout or Java’s System.out. These two methods cause text to be accumulated to the transcript. #cr writes a newline, and #show: writes its provided string.

A.7 Summary

This appendix has provided a very simple overview of the fundamentals of the Smalltalk language. More detail on the language can be found in the ‘Blue Book’, Smalltalk-80: The Language and Implementation [17].
Appendix B

Specification of the Apostle Language

This section contains the specification of the Apostle language. The declarative language is presented in Section B.1, and Section B.2 describes the Apostle MOP.

B.1 Declarative Specification

B.1.1 Pointcuts

A set of join points are identified through use of pointcut. Pointcuts are expressed in a compact, distinct language, closely resembling logic queries. They are built by composing a small set of primitive pointcuts into more complex queries using standard set operators: intersection ('&'), union (' |') and complement ('!'). The operators for intersection and union were chosen in keeping with Smalltalk’s logical AND and OR operators. Pointcuts are defined on and evaluated in the scope of a class or aspect.

Primitive Pointcuts

Apostle provides the following primitive pointcuts:

kindOf(type-signature)  Identify join points where the executing object is of the given type. The type-signature is of the form:

{namespace.type {class}

The namespace, if specified must be followed by a period. The join points on a class' class methods are identified with the class modifier. Both the namespace and type support standard Smalltalk wild-cards, where '#' matches any single character and '*' matches any sequence of characters. An unspecified namespace is treated as '*'. The instance can be bound for advice by providing a name, as discussed in Section B.1.5. (Corresponds to AspectJ's instanceof() designator)
receptions(selector) Identify join points where selector is received before being dispatched to the appropriate implementation. Or in English, identifies places where sending selector would not cause a #doesNotUnderstand. The arguments to the selector can be made available by providing them a name (Section B.1.5). (Corresponds to AspectJ’s reception() designator)

executions(selector) Identify join points where selector is executed. Or in English, the implementations of selector. The arguments to the selector can be made available by providing them a name (Section B.1.5). (Corresponds to AspectJ’s execution() designator)

cflow(pointcut-expression) Identifies points in the program’s execution occurring as a result of execution having passed through a join point identified by pointcut-expression. Cflow pointcuts cannot currently capture the context of the cflow location. (Corresponds to AspectJ’s cflow() designator)

sender(method-signature) Identifies those join points where the previous sender on the stack conforms to method-signature, specified as:

```
{ namespace. }type { class } { ^selector }
```

The type and arguments can be captured, as in(kindOf() and receptions()); discussed further in Section B.1.5.

nil As Smalltalk does not support any notion of ‘abstract’ program elements (i.e. not complete), we provide a nil pointcut. This is an all-exclusive pointcut: i.e. it matches nothing. Unknown named pointcuts are equivalent to nil.

B.1.2 Naming Pointcuts and Referencing Named Pointcuts

Pointcuts can be named, promoting information hiding by separating its definition from its use [39].

```
pointcut pointcutName(formally exported args): 
  pointcut-designators
```

Named pointcuts can be referenced from other pointcuts. For example:

```
pointcut foo (): kindOf(Aardvark) & (receptions(add:) | SpecialDictionary.additions())
pointcut bar (): cflow(foo()) & kindOf(Block) & receptions(value:)
```

B.1.3 Advice

Advice is defined on an aspect, and serves to attach code to some set of join points, identified using a pointcut. It can be seen as annotating the execution of the program. Advice takes
the form:

```xml
<advice-specification>
  <advice-body>
</advice-specification>
```

There are currently three forms of `<advice-specification>`:

**before `<pointcut-designator>`** Executed before any identified execution points. Short of causing an out-of-context return, before-advice cannot affect the identified execution point.

**after `<pointcut-designator>`** Executed after any identified execution points have occurred. Should the execution point be reached, the advice-body is guaranteed to execute. After-advice cannot affect return value of execution point; indeed, the advice may occur due to an out-of-context return, in which case there is no return value!

**around `<pointcut-designator>`** Executed around the identified execution point. The actual execution point is continued by using the `PROCEED` or `PROCEEDUSING` keywords; their omission prevents the execution of the original point and any subsequent advice. The return-value of the execution point is taken to be the return-value of the advice-body.

Apostle does not provide AspectJ’s additional specializations of after advice, after-throwing and after-returning; these exist to satisfy static typing requirements that are not required of Smalltalk.

**The Inheritance of Advice** Advice is not inherited in Apostle. This is an implementation detail at present, but made acceptable by the reasoning that advice bodies may be best used as bridging to other method bodies [12], which are inherited.

**B.1.4 Aspects**

Aspects are similar to classes, but instead embody a crosscutting concern, serving as the repository for the state across the concern. They serve are the definition point for advice, and define the scoping point for any referenced pointcuts used. Unlike classes, aspect instances cannot be created explicitly. Instances of an aspect are rather created upon the satisfaction of some per-aspect condition, and last for the duration of the condition. There are three types of aspects:

**Singleton** Only one single instance of this aspect will exist within the system, and the instance will last the existence of the system, like the Singleton design pattern [15]. These are similar to the ‘of eachJVM’ aspects in AspectJ.
**Per-Object** Per-object aspects are instantiated upon any execution transition to any object of a type matched by its configured pointcut. The instance exists for the lifetime of the object. These are similar to the ‘of eachobject()’ aspects in AspectJ.

**Per-Cflow** Per-cflow aspects are similar to per-object aspects, but the instance only lasts for the duration of the context. For example, if a per-cflow’s pointcut identifies `Dictionary^#at:put;`, then an instance will be created upon entering this method, and will last until this method is exited. These are similar to the ‘of eachcflow()’ aspects in AspectJ.

Aspects are built using Apostle’s MOP facilities (Section B.2.1, just as classes are constructed using Smalltalk’s MOP. An aspect’s type can differ from its superaspect’s.

### B.1.5 Capturing the Execution Context at a Join Point

Selected parts of the execution context at a join point can be made available to advice through name capturing. Several of the primitive pointcut designators support this, such as `kindOf()` and `type()`, `receptions()` and `executions()`, and `sender()`. This is done by specifying a name at the appropriate locations. Consider the following example: Capturing a method argument, for example, might be done like so:

```
before<kindOf(KeyedCollection) & receptions(#at: key put:)> aspect accessedKey: key
```

Captured names can be formally exported by named pointcuts, providing a greater degree of separation between the join points identified and the advice body. A rather contrived example of this might be:

```
pointcut fetches (dict, key):
(kindOf(Dictionary dict) &
 (receptions(at: key) | receptions(at: key ifAbsent:)))
| (kindOf(MtSmtpMessage class) & receptions(header: dict at: key))
```

where `Foo>>#header:at:` extracts the given key from the provided header. Apostle provides the appropriate binding of `dict` and `key`, regardless of whether the binding is the receiver or an argument.

Arguments cannot currently be captured in `cflow` pointcuts.

### Replacing Formally Declared Arguments

The formally bound arguments presented to `around` advice can be overridden or replaced using the special `PROCEEDUSING` instead of `PROCEED`. This is treated as a special form of keyword-selector, whose keywords must match one of the formally-bound arguments. Arguments not specified will continue with their original values. Continuing the `fetches()` pointcut defined just above:

```
around<fetches(dict, key)>
```
This advice causes ‘key’ to be replaced with the case-insensitive wrapper for any subsequent advice. It is able to do this without knowing anything about the actual join point, whether it be `Dictionary#at:put:` or `MtSmtpMessage class#header:at:`.

Only method arguments to the active join point can currently be replaced, as the semantics of replacing non-arguments is unclear. Consider what should happen in the following:

```smalltalk
around<fetches(dict,key)>

^PROCEEDUSING dict: Dictionary new
```

If the join point is `Dictionary#at:put:`, should the receiver be replaced? It may be possible in the future to replace any of the context values, but it silently fails at present.

## B.2 Apostle's MOP: A Programmatic Interface

Having seen the Apostle language and its declarative representation, we proceed to describe Apostle's programmatic interface. Given Smalltalk's reflective capabilities, such an interface must be provided.

### B.2.1 Aspects

Aspects are represented as subclasses of the `ApAspect` class. There are three types of aspect, as described in Section B.1.4. Notably, an aspect's type can differ from that of its superaspect; i.e. a per-cflow aspect can inherit from a singleton aspect.

Aspects are created by sending a creation message to the appropriate superaspect, similar to creating classes. These creation messages specify the actual aspect type. Singleton aspects are created by:

```smalltalk
superAspect
subaspect: #aspectVnameinstanceVariableNames:

\text{inst var names}''
\text{classVariableNames: 'class var names'poolDictionaries: '}
\text{inst var names}'
```

Per-Object aspects are created with:

```smalltalk
superAspect
subaspect: #aspectVnameperObject:
\$<\text{pcut-designator}$>$instanceVariableNames:
\text{inst var names}'
\text{classVariableNames: 'class var names'poolDictionaries: '}
\text{pool names}''
```

with an appropriate `pcut-designator`. PerCflow aspects are created very similarly to per-object aspects:
Each has a similar variant for declaring class instance variables, such as `#subaspect:class::instanceVariableNames:`. Once sent, these creation messages take immediate effect.

The aspect instance for an object can be obtained by providing the instance to `aspectClass>>#aspecfFor:`. `nil` should be used for aspect-types not directly bound to an object.

```smalltalk
Aspect aspectFor: self.
Aspect aspectFor: nil.
```

## B.2.2 Pointcuts

Pointcuts are implemented as subclasses of `ApPointcut`, and primitive pointcuts of `ApPrimitivePointcut`, itself a subclass of `ApPointcut`. An example of building a pointcut might be:

1. `(kindof := ApKindOfPrimitivePointcut namespaceName: 'CLDT') capturedName: 'cl'.`
2. `receps := ApReceptionsPrimitivePointcut selector: 'at: key'.`
3. `cflow := ApCflowPrimitivePointcut pointcut: (kindof & receps not)`

Pointcuts can be composed with others using boolean operators `\&` and `|`, and the complement `#not`. There are several kinds of primitive pointcut designators, described further in Section B.1.1.

As the `receps` pointcut shows in lines 1 and 2, names can be captured as would be specified in the declarative syntax. `kindOf()` and `type()` require explicitly notifying the captured name, as does `sender()` for its type name. Method arguments are specified, as in line 2, by interspersing the names in the appropriate position of the selector.

Pointcuts are generally defined on a class or aspect, specified with `#definer:`, defining its scope. If not specified, then any references to other pointcuts are unscoped.

### Naming Pointcuts

Pointcuts can be named by sending `#name:` with the appropriate name. Naming advice will reapply any advice using such named as advice; this is done by sending either `#apply` or `#remove` to the pointcut.

```smalltalk
|pcut|
pcut := ((ApKindOfPrimitivePointcut typeName: 'Bag') capturedTypeName: 'b') &
(ApReceptionsPrimitivePointcut selector: 'at: key').
```

```smalltalk
definer := Object;
name := 'fetches';
formallyExportedNames := #( 'key' );
apply
```
Named pointcuts can be referenced using `ApNamedPointcutReference` as in:

```plaintext
ApNamedPointcutReference
   referencing: 'PercolationSimulator.eventFired'
   usedNames: #('sim' 'event')
```

This allows the named pointcut’s definition to change.

### B.2.3 Advice

Advice is represented as instances of the different subclasses of `ApAdvice`, specifically `ApBeforeAdvice`, `ApAroundAdvice` and `ApAroundAdvice`. The join points to be annotated with this advice are identified by a provided pointcut. Advice bodies are provided as code fragments, and can use any dialect-specific extensions (e.g. Squeak’s curly-braces).

```plaintext
( advice := ApAdvice type: ApBeforeAdviceType )
   pointcut: pointcut;
   definer: SomeAspect;
   snippet: 'self isFull ifTrue: [self error: "is full"];
   advice apply.
```

The advice will be put into effect once sent the `#apply`, and removed with `#remove`.

### B.2.4 Batching Changes

Earlier work on implementing AOP languages added a ‘weaving’ step [30] to the compilation process, one that distributes the advice to the appropriate locations. Since there is no such compilation phase in Smalltalk, weaving is done at all times. More important is to remember where these changes have been woven; we record these changes on a loom.

Looms are represented as instances of the class `ApLoom`. There is generally one per system, accessible by `ApLoom current`, which records all changes made to the system. Looms are not normally manipulated by programmers; all MOP objects provide direct protocol for their respective actions.

Groups of work can be batched, or represented declaratively, by applying a roster of work to a loom. These rosters are represented by instances of `ApWorkRoster`, allocated by sending `#newWorkRoster` to an appropriate loom. Rosters support:

- adding, renaming or removing pointcuts;
- adding or removing advice;
- adding, replacing, or removing new program elements;
- adding, changing, or deleting a class.
Once described, they are applied to their loom with \#apply. The respective \#apply or \#re-
move operations of the various Apostle MOP objects are themselves implemented through 
this work-roster interface.

B.2.5 Reflective Capabilities

Executing advice is provided a special object named thisJoinPoint encapsulating its current 
execution location. It can be queried for the object’s type, and the selector name.

Classes and aspects can be queried for their defined advice and pointcuts. I.e. 
ApAspect class\#advice, Class class\#pointcuts. ApAspect class\#pointcuts. The 
loom can be queried for current advice.