Refactoring Aspect-Oriented Software

by

Shimon Rura

A Thesis
Submitted in partial fulfillment of
of the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 23, 2003

Abstract

This thesis extends the state of the art in refactoring to Aspect-Oriented programming. Refactorings are specific code transformations that improve the design of existing code without changing its observable behavior. Aspect-Oriented Programming (AOP) offers a new approach to software design by encapsulating crosscutting concerns. The novel contributions of this thesis are a recasting of existing refactorings to preserve program behavior in aspect-oriented code, and several new refactorings that can improve the design of code by deploying AOP techniques. The refactorings are described in reference to AspectJ, an AOP extension of Java, and are amenable to partial or full automation.

It is necessary to reevaluate existing OO refactorings because the constructs of AOP programming languages significantly affect what changes can be meaning-preserving. To this end, new preconditions and steps are introduced to about 20 fundamental refactorings (from [Opd92]) such as renaming a class and inlining a method. These extended refactorings can form the basis for an AOP-aware refactoring tool.

About thirty new AOP-specific refactorings are proposed. These refactorings include both fundamental refactorings and more complex refactorings built from these that address specific design problems. For example, a simple refactoring possible in AOP is to move the definition of a method from within a class to an aspect. A more complex refactoring that includes this is moving all code responsible for implementing a particular interface into an aspect. The focus is primarily on accomplishing the desired changes once the involved program parts are identified. These new refactorings can form the basis for a truly AOP-focused refactoring tool.

Acknowledgements

None of this work, the big abstract thinking or the oodles of writing, would have been possible without my advisor, Barbara Lerner. Since I started working with Barbara, I've done some of the most challenging and fun work I can recall. I have started to understand and enjoy the peculiarities of research. For all your persistent attention, your flexibility, and your skill in prodding me to work hard, Barbara, thank you.

Thanks are also due to Kim Bruce, my second reader. I could hardly wish for a more responsive, insightful, or conscientious reader—especially for a subject so closely related to programming languages. I had the privilege of studying Programming Languages with Kim as my instructor, an experience that developed my existing interest and provided me with a thorough foundation in the discipline.

This work builds on a number of different sources, too many to give adequate thanks to them all. But foremost among these are the Ph.D. thesis work of Williams F. Opdyke and the AspectJ language and tools developed at the Palo Alto Research Center and other organizations. The quality and relevance of their work inspired and guided much of mine.

Thanks to all of my dear friends who have amused, supported, and loved me this year. Special thanks my lab and dorm mate Josh Ain, with whom I always get along well. Thanks to my friends in CS and Williams Students Online, especially Brent "the cool" Yorgey, Tom "suddenly hacking the WSO site" White, Jesse "Pants?" Dill, Evan "54 and light rain" Miller, Chris "the $6000 man" Douglas, Steve Winslow, Robin Stewart, Chris Cyll, Ben Cohen, and Dan Weintraub. And thanks to my closest friends Jenny, David, Beth, Ben, and my mom Lyenna Rura for always being there for me.

And finally, thanks to the wonderful faculty and staff in the Computer Science department at Williams College. I know and admire all of you, and I can't imagine a finer group of people to study with. Thanks to Andrea Danyluk for advising me early on, teaching me in 3 excellent courses, and helping to keep the department cheerful. Thanks to Bill Lenhart for being an awesome prof and understanding what thesis crunch time means. Thanks to Duane Bailey for encouraging myself and other students to work hard, to have fun, and to take pride in our work. Thanks to Steve Freund for having a ponytail and riding bikes. Thanks to Tom Murtagh for teaching me all that stuff in Compilers that ended up helping me be able to do this thesis. Thanks to Jim Teresco for giving Josh *something* to do, or else I'd still be laughing too hard to get anything done. Thanks to Mary Bailey for going out of your way to help me out and cheer me up day after day in lab. Thanks to Lorraine Robinson for being amazingly generous of kindness, office supplies, and snacks. I'll miss you all.

Contents

Chapter 1

Introduction

"Design is hard. . . . Reusable software usually is the result of many design iterations."
–William F. Opdyke, Ph.D. Thesis

This thesis extends the state of the art in refactoring to aspect-oriented software by defining correct and effective refactorings that operate on programs written in an aspect-oriented programming language. Refactorings are specific code transformations that improve the design of existing code without changing its observable behavior. Aspect-Oriented Programming (AOP) offers a new approach to software design by encapsulating crosscutting concerns. Though both of these approaches are concerned with enabling better design, existing work in refactoring is generally inapplicable to AOP systems. This is because the program constructs available in AOP languages can affect program behavior in ways that make many existing refactorings non-behavior-preserving. Furthermore, as aspect-oriented programming becomes more commonly used for real software systems, there is a growing need to understand refactorings that will help programmers take advantage of aspect-oriented features. This thesis works toward the two goals of extending existing refactorings for AOP and developing new, AOP-specific refactorings.

1.1 The Problem

Aspect-oriented programming languages augment object-oriented languages with constructs that can alter program structure and behavior across module boundaries. When refactoring a program that contains these constructs, it may not be sufficient to apply refactorings that only analyze and transform object-oriented parts of the code—AOP language features can change inheritance hierarchies, add members to classes, or invoke code when certain well-defined points in program execution are reached. For example, consider renaming a method `setX(int)` in a class `Foo`. In pure OO code, it is enough to check that the new name will not conflict with another method in `Foo` and fix all references (calls) to `setX(int)`. However, in the AspectJ language (an extension of Java), we might write some code that depends on `setX` as follows:

```
after() : call( Foo.setX(..) ) {  notifyObservers(); }
```

This is a line of *advice* that will trigger a code block, here `notifyObservers()`, after the completion of every call to any method in `Foo` called `setX`. If we rename `setX` without adjusting the *method pattern* we use here to match its signature, this advice will no longer be triggered, `notifyObservers()` will not be executed after calls to the method, and program behavior will not be preserved. To be able to apply refactorings like this correctly, we must determine what kind of extra work is needed to make each refactoring behavior preserving in aspect-oriented programs.

A problem separate from but related to fixing OO refactorings for AOP is setting down new, AOP-specific refactorings. AOP's greatest strength is in encapsulating crosscutting concerns. But if the implementation of a concern is already spread throughout our software, how do we abstract those scattered pieces into an aspect while making certain not to change what our program does? This challenge calls for a set of new, AOP-specific refactorings.

1.2 Characteristics of a Solution

Updating Object-Oriented Refactorings. We must show how known refactorings can be made behavior-preserving when applied in a specific AOP programming language. There are several works that describe refactorings (see section 1.4.1), each with slightly different goals like developing automated refactoring tools [Opd92] or helping practicing programmers familiarize themselves with specific ways to improve design [Fow00]. Of these, this thesis builds primarily on William Opdyke's Ph.D. thesis work [Opd92], the first to consider refactoring in object-oriented software. Opdyke distills the more complex OO refactorings down to iterative applications of simpler, *fundamental* refactorings. This approach is particularly well suited as a basis for extending refactorings to AOP, because if the fundamental refactorings can be adapted, the higher-level refactorings are trivially adaptable. Because the approach is tool-oriented, it provides a useful minimum set of refactorings from which many OO refactorings can be assembled.

Defining New AOP-Specific Refactorings. A set of new, AOP-specific refactorings must be described. There are many ways to discover such refactorings, the most promising of which is by analysis of design evolution in long-lived software systems. Unfortunately, long-lived systems programmed in aspect-oriented languages are rare or nonexistent. Therefore, the new refactorings described in this thesis are derived from comparisons of AOP and OO designs [HK02] and the author's intuition. This approach lends itself well to discovering fundamental refactorings, but not as well to more specific higher-level refactorings. All of the refactorings are described with clear dependencies on the analyses and transformations required to make them behavior-preserving.

1.3 Organization of this Thesis

The following two sections give a more extensive background on refactoring and aspect-oriented programming, and this chapter concludes with a discussion of contributions of this thesis.

Chapter 2, Requirements for Behavior Preservation, explains what is meant by behavior preservation, and establishes a set of constraints that, if satisfied by a particular program transformation, guarantee that it is a refactoring. It also describes characteristics of programs that make it hard or impossible to ascertain that a refactoring is behavior-preserving.

Chapter 3, Low-Level Refactorings Reconsidered, reviews the set of low-level refactorings defined by Opdyke in [Opd92], giving new preconditions and transformation steps for each and providing brief arguments that each modified refactoring is behavior-preserving in AOP based on the constraints in Chapter 2.

Chapter 4, Working with Patterns, describes methods for analyzing and transforming patterns, AspectJ's lowest-level ways of referring to program elements.

Chapter 5, New Fundamental Refactorings for AOP, defines about twenty AOP-specific fundamental refactorings.

Chapter 6, High-Level AOP Refactorings, describes how the fundamental refactorings can be combined into more specialized refactorings, including three example high-level refactorings designed to abstract crosscutting concerns using AOP techniques.

Chapter 7, Conclusions, summarizes contributions and limitations of this thesis and points to several areas for future work.

Appendix A reproduces a quick reference card for the AspectJ language.

Appendix B gives a grammar for features specific to the AspectJ language.

1.4 Refactoring

Refactoring is the process of changing the design of existing software—modifying the structure of its code without affecting its external behavior. A variety of specific *refactorings* might be employed in this process. Each refactoring describes a disciplined way to modify code in order to achieve a specific design change, without introducing bugs, leaving the code in a less complete state, or otherwise tampering with the way the software behaves. For example, one might refactor to replace direct uses of public fields with accessors and modifiers, or create an abstract superclass to encapsulate common behavior in similar classes.

The possibility of changing design after code has been written enables a more flexible notion of software development, instead of the elusive waterfall approach where a clear understanding of the problem at hand leads to an elegant design that is finally implemented in code. In real systems, even if we could come up with the right design at first, there is virtually always a need to change the software throughout its lifetime.

When it comes time to make a change, the obvious approach is to implement the new concern in the way that demands minimal alterations of existing code. This reduces the risk of breaking existing behavior in the process. Its long term consequences, however, depend on how the minimal change fits into the system's design: if the design has anticipated the right kind of change, then it is simply a matter of filling in the detail in the right place; the integrity of the design is just as strong as it was before the implementation, and the change is localized and unpluggable. On the other hand, if the new concern cannot be elegantly accommodated, the obvious implementation is likely to be one that selectively compromises the design in many, scattered places. This makes the new implementation hard to maintain; its parts may live anywhere in the code base and depend on numerous implicit assumptions. When many of these changes get hacked onto a software system, it becomes hard to trust that any single part can be understood on its own, as a module. Consequently, the design begins to lose value, and the cost of maintenance grows and grows.

With refactoring it is possible to avoid this downward spiral. When the design of existing code does not accommodate needed changes, it can be reworked in-place so that it will. The design can evolve along with the new requirements. Though the short-term cost to implement a particular change will be higher than that of the simplest hack, the long-term benefits may be so great that the added cost is justified. Advocates of refactoring argue that it opens the door to disciplined, piecemeal growth of software systems [Fow00].

Aside from preparing software for extensions, there are other motivations for refactoring. Opdyke [Opd92] describes the following:

Extracting a reusable component. Consider a business system that has been in place for several years. It may become desirable to replace the underlying control of business operations with an industry-standard system when one is made available, but to preserve the existing system's user interface so that users need not be retrained. In the existing system, user interface functions are interwoven with other, obsolete functions. The software can be refactored so that the user interface component can be extracted.

Improving consistency among components. Different components may be implemented by different project team members. A pair of components initially thought distinct may turn out to share some common substructure. To make these components easier to understand and to improve the maintainability of the system, the components can be refactored to make their common elements explicit.

1.4.1 The Discipline of Refactoring

Programmers have likely been applying *ad hoc* meaning-preserving program transformations since the advent of high-level languages, but refactoring was first studied formally in the early 1990s [OJ90, Gri91, Opd92, GN93]. Recently, refactoring has gained momentum, partly because of its prominence in cutting-edge software development methodologies such as Extreme Programming [Bec99]. Several advances in refactoring have included development of refactoring tools for specific languages; first Smalltalk [RBJ97] and, more recently, Java, C++, and others [Fow02, Goo02].

William Opdyke's PhD thesis [Opd92, OJ90], under the direction of Ralph Johnson, was the first major written work using the term *refactoring.* He considered refactorings for object-oriented software, identifying a set of refactorings applicable in that context. For these refactorings, he described the design prerequisites and automatic program restructurings required to guarantee preservation of behavior. Though his experiments were based on C++, an important consequence of Opdyke's work was the later development of a refactoring tool for Smalltalk. This tool, the Refactoring Browser [RBJ97], was also developed under the leadership of Ralph Johnson at the University of Illinois, and was implemented by John Brant and Don Roberts.

Other significant early work by William Griswold explored a variety of refactorings (though he did not use this term) and related transformations [Gri91, GN93]. Like Opdyke, Griswold focused on separating the process of refactoring from other software modifications, and described a set of automatable transformations in terms of their requirements and effects. Griswold's work, however, was primarily concerned with functional and imperative programming, and included a prototype tool for restructuring Scheme programs.

In recent times, the surging popularity of refactoring in industry has led to the publication in 2000 of a more practically-oriented book, by Martin Fowler [Fow00]. Fowler presents a refactoring guide for professional programmers, including questions of how refactoring figures into the software development process, along with an extensive catalog of object-oriented refactorings with examples in Java. It describes over 70 refactorings, from very simple (such as *rename method*) to quite complex (such as *tease apart inheritance*[1]).

Fowler's book is a sign of booming interest in the techniques of refactoring. This has been fueled especially by the rise of Agile Software Methodologies [B+01] such as Extreme Programming (XP) [Bec99]. These methodologies emphasize an incremental development process, with little up-front design and short or virtually continuous development iterations. At each iteration, the software may not address the complete set of requirements, but carries a subset or simplification of the requirements through the development cycle, from design to release. Each iteration is expected to be designed to appropriately suit the current set of features; as future iterations add new requirements,

[1] A single inheritance hierarchy doing two jobs at once invites code duplication and complicates extending the hierarchy. Because separate concerns have interwoven implementations, making one small change may require a bunch of parallel changes. The purpose of *tease apart inheritance* is to create a separate class hierarchy for each secondary job and delegate to it where appropriate.

the software is refactored to preserve an appropriate design. In this context, refactoring is frequent and crucial. Historically, the concepts of refactoring may have inspired and enabled agile methodologies: Kent Beck, creator of XP, worked extensively in Smalltalk with the Smalltalk Refactoring Browser [RBJ97] for years before he described refactoring, along with other development strategies, in a software development methodology.

1.4.2 The Rise of Refactoring Tools

When refactoring is done frequently, a tool that could automatically apply refactorings—quickly and without making mistakes—could significantly reduce development time and costs. The first tool designed to assist with refactorings on a large scale was the Smalltalk Refactoring Browser [RBJ97]. It automates many refactorings including renaming variables, methods, and classes; extracting new variables, methods, and classes to encapsulate concerns in existing code; inlining methods and classes to remove unnecessary abstraction; and moving components between classes and up or down the inheritance hierarchy.

Because the Smalltalk Refactoring Browser was designed to "bring refactoring into the mainstream of program development" [RBJ97], it models some important features for refactoring tools. It is integrated with standard development tools, fast, and careful to prompt the user for names when appropriate. In these ways, it achieves results that can be tried and, if necessary, undone; and does not insert meaningless identifiers into code.

Due to the dynamic type system of Smalltalk, some of the browser's refactorings have rather complex implementations. For example, in order to rename a method, each call site must be modified to reference the new name. Of course, prior to the refactoring, another type may have a method call of the same name; its calls should not be changed. Unfortunately, because the type of an object often cannot be known statically, there are cases where this refactoring cannot be automatically performed statically. Thus in the Smalltalk Refactoring Browser, rename method is a *dynamic refactoring:* part of its execution requires exercising the program with a test suite.

This exact issue does not occur in Java or other statically typed languages: many refactorings can be accomplished safely with much simpler analyses. Though there may be refactorings for statically typed languages that would require dynamic analysis, I am not aware of any work toward identifying or automating such refactorings.

A number of tools support refactorings in Java, including specialized refactoring software as well as general-purpose interactive development environments (IDEs). The IDEs include commercial ones such as IntelliJ IDEA (`http://www.intellij.com/idea/`) and the open-source Eclipse IDE (`http://www.eclipse.org/`). Dedicated tools include Xrefactory (`http://www.xref-tech.com/speller/`), a high-performance code browser and refactoring tool for Java and C++; JREFactory (`http://jrefactory.sourceforge.net/`) and Transmogrify (`http://transmogrify.sourceforge.net/`) for Java, and others. (Current lists are available at [Fow02] and [Goo02].)

While these tools vary considerably in their implementation and styles of usage, they generally support the same kinds of refactorings:

Rename: semantics-preserving renamings of variable, method, and class identifiers.

Move: relocate fields, methods, and classes up or down the class hierarchy or into other classes and packages, fixing references as necessary.

Extraction: select a section of code to abstracted into a method or an expression to be abstracted into a variable. The best example is *Extract Method,* which encapsulates a code selection in a method declaration, determines which values must be passed in as parameters, and searches the program for equivalent code to be replaced with a method call.

Inlining: the opposite of extraction. Replaces references to method calls or variables with an in-place procedure or expression.

1.4.3 A Refactoring Example

Fowler [Fow00, p.135] describes the refactoring *Replace Method with Method Object,* designed to enable decomposition of a long method that depends on many local variables. The goal is to turn the method into its own class, so that all the local variables become fields of the new class[2]. Once this is done, the real benefits of decomposing the long method can be achieved using the *Extract Method* refactoring inside of the new class. Figure 1.1 shows example code before refactoring.

```
class Order...
  double price(Vector extras) {
    double basePrice = getQuantity() * myCost;
    double extrasPrice = extras.size() * extrasCost;
    double servicePrice = service.getPrice(yearsServiced) +
                          service.getPriceForExtras(extras.size());
    double deliveryPrice = ...    // and so on...

    return basePrice + extrasPrice + servicePrice +
      (isDelivered? deliveryPrice : 0);
  }
```

Figure 1.1: A long method before the refactoring.

First we create a new class, named after the method. We give it fields for the source object and each of the method's local variables and parameters:

```
class PriceCalculator {
  private final Order _order;
  private Vector extras;
```

[2]The *Method Object* pattern is from [Bec97].

```
  private double basePrice, extrasPrice, servicePrice, deliveryPrice;
}
```

To this we'll add a constructor, taking the source object and the parameters:

```
PriceCalculator(Order o, Vector e) { _order = o; extras = e; }
```

Now we'll add a `compute()` method to the new class, moving the body of the original method over. In the process, we need to change references to members of the Order object to go through `_order`:

```
double compute() {
  basePrice = _order.getQuantity() * _order.myCost;
  extrasPrice = extras.size() * _order.extrasCost;
  servicePrice = _order.service.getPrice(_order.yearsServiced) +
                 _order.service.getPriceForExtras(extras.size());
  deliveryPrice = ...    // and so on...

  return basePrice + extrasPrice + servicePrice +
    (_order.isDelivered? deliveryPrice : 0);
}
```

And finally, to invoke our new code, we'll put the class declaration inside of the `price()` method, and have it simply construct a PriceCalculator and then delegate to its `compute()` method. The final code is shown in figure 1.2.

Analysis

Unlike many refactorings, all the changes that *Replace Method with Method Object* requires are localized within the space of a single method, eliminating the need for a global reference search—usually among the most compelling reasons for automated assistance. However, this refactoring still exemplifies an opportunity for tool support. Its steps could be fully automated based on the selected `price()` method:

1. **Creating a new class.** A class is created, inner to `price()`. It is given instance variables for the object containing `price()`, that method's parameters, and that method's local variables.

    - The source object is declared `final` to enforce some preservation of semantics; assigning to `_order` would be like assigning to `this` in the Order object, and so should not be allowed.
    - To keep this refactoring simple to complete, all locals are treated uniformly. In later refactorings, some of the local variables that have become instance variables may become localized.

2. **Constructor creation.** The new class's constructor is a trivial one that assigns the variables for the source object (`Order`) and the method parameters.

3. **Moving the method body.** The body of the `price()` method is moved and transformed into a new `compute()` method.

   - As local variables have been replaced with instance variables, their declarations in the method body must be removed. Other references to these variables need not be changed.
   - References to instance variables and method calls on the source object need to be replaced with indirection through the `_order` field.

4. **Change the original method to use the method object.** Replace the original method body with construction of the method object and delegation to its `compute()` method.

Each of these steps can be fully automated, with the exception of naming the new class and method. In addition to accelerating the refactoring process, tool support guarantees that no mistakes will creep in, and that behavior is preserved—the latter is particularly

```
class Order...
  double price(Vector extras) {
    class PriceCalculator {
      private final Order _order;
      private Vector extras;
      private double basePrice, extrasPrice, servicePrice, deliveryPrice;

      PriceCalculator(Order order, Vector extras) { ... }

      double compute() {
        basePrice = _order.getQuantity() * _order.myCost;
        extrasPrice = extras.size() * _extrasCost;
        servicePrice = _order.service.getPrice(_order.yearsServiced) +
                       _order.service.getPriceForExtras(extras.size());
        deliveryPrice = ...    // and so on...

        return basePrice + extrasPrice + servicePrice +
          (_order.isDelivered? deliveryPrice : 0);
      }
    }

    return new PriceCalculator(this, extras).compute();
  }
}
```

Figure 1.2: The completed *Method Object* refactoring on the `price()` method.

easy in this case, where the interface of `Order` is unchanged as all refactoring occurs within the `price()` method. Even barring mistakes, the application of this refactoring may be tedious, discouraging its use. By making its application virtually effortless, programmers may be more inclined to use this refactoring where appropriate.

1.5 Aspect-Oriented Programming

Aspect-Oriented Programming, or AOP, extends Object-Oriented Programming with the concept of *aspects,* which modularize crosscutting concerns. Like a class, an aspect is intended to capture a set of related program elements addressing a particular concern. Unlike classes, however, aspects are intended to modularize *crosscutting* concerns—those that inherently span the definitions of many classes. AOP techniques let the programmer specify well-defined ways that aspect code blends with other program code. Depending on the AOP mechanism in use, these effects may be static or dynamic, altering the software's structure or execution.

1.5.1 The Problem of Crosscutting Code

A primary goal of object-oriented programming is to enable programmers to design code whose structure mirrors the real, live structure of the objects being modeled. This is achieved by describing the states and operations that may apply to classes of objects. But many software systems must address concerns that are not localized to a single class. For example:

Error handling: many classes comprise a service, each implementing error handling/reporting at relevant points in its operation. Even if these chunks of error-handling code are initially implemented in a coordinated way, it will be difficult to understand or change the error reporting behavior and structure, because it is intertwined in unpredictable ways with code pertaining to the class's primary functions.

Caching: because caching is a performance concern separate from functional concerns, intertwining caching code with other code can complicate both. Additionally, while it may be easier to achieve an optimal caching policy by taking into account a variety of different classes of data objects, spreading this knowledge throughout those classes may introduce couplings and interdependencies between functionally unrelated classes.

Multi-object protocols: some protocols and design patterns, such as the Subject-Observer pattern [GHJV95, pp. 293-303], assign roles to multiple objects. Each object may need to implement new code that does not relate to its functional purpose solely to participate in the protocol. Again, pure OO design forces independent concerns to be bound together at design time.

These examples illustrate *crosscutting* concerns, and highlight the limitations of even the best object-oriented designs in adequately modularizing these concerns. With pure

OO approaches, the implementation parts of these concerns will be mixed in throughout other code. Though inheritance or other techniques may mitigate these problems in some cases, AOP seeks to directly model the crosscutting structure. Thus aspects are comprised of code along with explicit specifications of where and how that code joins with the rest of the program.

Example
Memoization

For a specific example, consider the case of function memoization[a]: saving the results of a function in a table, so that if the function is invoked repeatedly with the same arguments, the value is simply looked up instead of computed [Mic68]. This is a performance enhancement that might be used in number-crunching systems. Because memoization has no effect on how the function is actually computed, we'd like to keep the computation separate from the memoization. But because memoization is a performance enhancement that must not be apparent to the function's caller (unless they are keeping track of time), we cannot change the interface. Ideally, we'd like to list which methods should be memoized, describe a generic memoization procedure, and have that procedure executed *around* calls to those methods. This is the approach we will pursue and refine in following examples. We'll start with the slow method defined below.

[a] I am indebted to Mark-Jason Dominus for first showing me the idea of memoization during his talk "Tricks of the Perl Wizards" given for the Cincinnati and Dayton, Ohio Perl Mongers group in 1999. His technique, which I believe can legitimately be called aspect-oriented, worked by redefining subroutines in strange ways (legal in Perl). See: `http://search.cpan.org/author/MJD/Memoize-1.01/Memoize.pm`

```
public static int myFunction(int x) {
    try { Thread.sleep(1500); } catch(InterruptedException e) { }
    return x * 3;
}
```

1.5.2 Language Mechanisms for Capturing Crosscutting Concerns

Like OO, AOP techniques can be achieved with a variety of mechanisms (a list of current software tools is at [Com02]). For AOP, these include preprocessors, class loader/manglers, component frameworks [PSDF01], and programming languages. The programming languages that provide the most robust AOP support each build on a popular OO or procedural language, currently including Java, C, C++, Smalltalk, Python, and Ruby. Of these languages, the most mature is AspectJ [KHH+01], an AOP extension to Java.

This section describes the primary language mechanisms that AOP languages use to capture and represent crosscutting concerns. Though the terminology comes from AspectJ [Tea02b], analogous mechanisms exist in other AOP languages such as Hyper/J [TOHJ99].

Static Introduction

In order to allow aspects to modify classes and their hierarchy, aspects may include several forms of *introduction*, which declares new members on classes (inter-type declaration) or alters inheritance relationships between classes. Introduction is based on the notion of *open classes*, and includes addition of fields and methods and declaration of superclasses[3] and implemented interfaces.

Inter-type declarations take hold at compile-time and can vary in visibility (e.g. private to the aspect or publicly visible). The introduction mechanism is important to AOP because it allows the many static parts of a crosscutting concern to be described in one place, even when the declarations must apply to a variety of separate and unrelated classes.

Example
Memoization

Since `myFunction` is static, we don't need to introduce its memo table into its containing class—we'll just keep it private to the Memoization aspect. A more flexible implementation might call for introducing fields to keep memoization tables for specific objects.

Affecting Program Behavior Dynamically

In addition to static introduction, aspects can affect the execution of the program dynamically. The language's *joinpoint model* specifies which joinpoints—well-defined points in the program's execution—can be described[4]. Based on joinpoint specifications, code contained in an aspect can be invoked during execution and affect behavior at runtime. The joinpoint model may vary considerably between languages; it could contain only very simple program events or allow elaborate (and potentially confusing) joinpoints based on less stable information such as variable names and line numbers. The most stable but expressive joinpoints are parameterized with class, method, or field names or name patterns, and include:

- method calls,
- method executions (when the actual method body is executed),
- object construction,

[3]Introduced inheritance declarations under languages supporting only single inheritance will only be legal if the newly declared superclass is a subclass of the original superclass.

[4]A *joinpoint,* like a *break point,* identifies a specific point in execution; but unlike break points used to stop a program during debugging, joinpoints invoke special code that can alter execution in a number of ways.

- class (static) initialization,
- reading and assignment of data in fields, and
- handling or throwing exceptions.

The code that specifies how program behavior is to be affected at runtime is *advice*. Advice has a great deal of power to inspect program state at runtime using reflection, and to manipulate state and execution paths. Advice might take hold

before a joinpoint: advice can view and modify input values and other state before the joinpoint is entered.

after a joinpoint: advice can view and modify return values and other state after a joinpoint has finished. There are also special cases of after advice for methods returning normally or exiting by throwing an exception.

around advice replaces the joinpoint. It can view and modify input, invoke the actual joinpoint using a special keyword, and view and modify its results. It is the only kind of advice that must declare a return type, and this must be a subtype of the joinpoint's return type. For example, around advice on a method call might look at the input parameters, and only delegate that information to the advised method if the parameters are within reasonable bounds. If the inputs are found bogus, it might trigger some recovery behavior and not call the method at all, perhaps returning an error value or not returning at all but throwing an exception.

Around advice only makes sense for joinpoints that evaluate to a value[5] (including a `void` value); the advice code is responsible for returning that value (possibly by invoking the actual code).

Example
Memoization

We'll use around advice to capture calls to the function we want memoized, look up their parameters in a table, and only delegate to the function when those parameters are not found.

1.5.3 AspectJ

AspectJ [Tea02b] is an extension of the Java programming language [GJSB00] with Aspect-Oriented additions. It is developed, freely distributed, and supported by researchers at the Xerox Palo Alto Research Center as a real-world test bed for Aspect-Oriented Programming. Because its developers want to test AspectJ on real projects in industry, it benefits from a stable core of support, including the considerable advantages

[5]This includes field assignments and accesses; in assignment, the new value is like a parameter to an assignment method, and the value of the assignment expression or field access is like the return value of a method.

of a mature, well-performing compiler; support in a number of interactive development environments; and active mailing lists for user support.

Any Java program is a valid AspectJ program. In addition, AspectJ programs may include structures called *aspects.* An aspect can include methods and field declarations like a class, but can also include *pointcuts,* which are sets of joinpoints; *advice,* which describes ways to alter program execution at joinpoints; and *introduction*, which transforms the static structure of a program.

Joinpoints and Pointcuts

AspectJ provides about 20 primitive pointcuts which select joinpoints based on names and properties. Pointcuts may be combined using set operators such as

`a && b`	all joinpoints in both `a` and `b`
`a || b`	any joinpoint in `a` or `b`
`a && !b`	any joinpoint in `a`, except those in `b`

The primitive pointcuts are parameterized by patterns that pick out sets of methods, constructors, types, or fields. In these patterns, `*` represents any sequence of characters; `..` in an identifier represents any sequence of characters starting and ending with '`.`'; and `..` in a series of parameters represents any number of parameters. For example,

```
execution(org.bar..* Foo.*(..)  throws My*Exception)
```

matches the execution of any method that throws MyFirstException, MySecondException, etc. The method may have any number of arguments, but must be in a class Foo, and must have a return type that is in some package whose name starts with "`org.bar.`". The following examples (mostly adapted from the AspectJ Quick Reference [Tea03]) demonstrate the basic pointcuts:

`call(void Foo.m(int))`: a call to the method with signature `void m(int)` in Foo

`execution(!public Foo.new(..))`: the execution of any non-public constructor of Foo

`initialization(Foo.new(int))`: the initialization of any Foo object that is constructed with `Foo(int)`

`staticinitialization(Foo)`: when the type Foo is initialized, after loading

`get(int Point.x)`: when the integer field x in the Point class is read

`set(!private * Point.*)`: when any non-private field in the Point class is assigned

`handler(IOException+)`: when an IOException or its subtype is handled with a catch block (without the `+`, type patterns match only an exact type)

The following primitive pointcuts can also be used on their own, but are commonly combined with the basic pointcuts in order to narrow down the range of included joinpoints. For example, a call-tracing aspect might include

```
call(* *(..)) && !within(Tracing)
```

which matches every method call except those within the `Tracing` aspect itself. The general forms[6] are:

`within(`*TypePattern*`)`**:** any joinpoint whose associated code is defined in any type matching the pattern

`withincode(`*MethodPattern*`)`**:** any joinpoint whose associated code is defined in a matching method

`withincode(`*ConstructorPattern*`)`**:** any joinpoint whose associated code is defined in a matching constructor

There are also primitive pointcuts that are parameterized by other pointcuts. These select points based on the flow of control in the parameter's joinpoints. For example:

`cflow( call(void Figure.move()) ):` any joinpoint in the control flow of each call to `void Figure.move()`, including the call itself.

`cflowbelow( call(void Figure.move()) ):` any joinpoint in the control flow of each call to `void Figure.move()`, not including the call itself.

Finally, a few pointcuts express additional conditions:

`if( Tracing.isEnabled() ):` any joinpoint where `Tracing.isEnabled()` is true. The boolean expression used can only access static members, variables bound in the same pointcut, and the reflective `thisJoinPoint` object.

`this( Point || Line):` any joinpoint where the currently executing object is an instance of Point or Line

`target( java.io.InputPort ):` any joinpoint where the target object is an instance of `java.io.InputPort`

`args( Point, int ):` any joinpoint where there are two arguments, the first a `Point` and the second an `int`.

`args( *, int ):` any joinpoint where there are two arguments, the second an `int`.

`args( float, .., float ):` any joinpoint where there are at least two arguments, the first and last of which are `float`s.

[6] A grammar for the patterns used to parameterize these pointcuts is given in Appendix B, p. 77.

Example
Memoization

Since our advice must apply around each call to `myFunction`, we will use the following pointcut, named `memoizedFunction`:

```
pointcut memoizedFunction(int x):
call(static int MyClass.myFunction(int)) && args(x);
```

This is an example of a named, parameterized pointcut: when the pointcut is used, a variable `x` must be passed in and its value will be bound to the `int` argument of `myFunction`. We will later reference this from within the advice code in order to look up or enter values in our memo table.

Advice

AspectJ allows advice to take hold before, after, and around joinpoints. Advice declarations can include formal parameters, which are passed to pointcuts and bound to values in joinpoints. The body of each advice declaration is executed at the appropriate time relative to each joinpoint. Although not all advice makes sense for every pointcut, current AspectJ syntax allows all combinations; in some cases, around advice simply behaves like before and/or after advice, depending on when and if the original joinpoint is invoked.

`before(): get(int Foo.y) {...}`: runs before reading the integer field `Foo.y`

`after() returning:` *pointcut* `{...}`: runs after the joinpoint returns normally

`after() returning(int x):` *pointcut* `{...}` runs after the joinpoint returns. The joinpoint must return an integer value. The value is bound to `x` in the body.

`after() throwing:` *pointcut* `{...}`: runs after the joinpoint throws an exception

`after() throwing(MyException e):` *pointcut* `{...}`: runs after the joinpoint throws MyException; the exception is bound to `e` in the body.

`after():` *pointcut* `{...}`: runs after the pointcut, regardless of how it returned

`before(int i): set(int Point.x) && args(i) {...}`: runs before the field `x` in any Point is assigned. The value to be assigned is named `i` in the body.

`before(Object o): call(void Vector.add(Object)) && args(o) {...}`: runs before calls to Vector's `add` method. The object to be added is named `o` in the body.

`int around(): call(int Point.getX()) {...}`: runs instead of calls to Point's `int getX()` method. The `getX()` may be invoked in the body using `proceed()`, which has the same signature as the around advice. Around advice may also declare thrown exceptions; these must not break Java's static type safety rules.

Example Memoization

We'll advise *around* our `memoizedFunction` pointcut and look up the argument in our memo table (the `memo` object here) to check whether `myFunction` really needs to run:

```
int around(int arg):  memoizedFunction(arg) {
   if(memo.hasValueFor(arg)) {
      return memo.getValueFor(arg);
   } else {
      int result = proceed(arg);
      memo.putValue(arg, result);
      return result;
   }
}
```

Weaving[a] an aspect containing the `memoizedFunction` pointcut and this advice together with the class containing `myFunction` will result in a program with memoization behavior on `myFunction`.

Full code of the memoization aspect follows.

[a]The process of activating aspects into other code is called *weaving*. Currently, the AspectJ compiler does all weaving at compile-time on source code or existing, unwoven class files; future releases of the AspectJ tools are planned to support weaving into already-compiled bytecode and weaving at class-load time.

```
public aspect MyFunctionMemoization {

  protected MemoTable memo = new MemoTable();
  // MemoTable provides:
  //  boolean hasValueFor(int argument), true iff argument is in table
  //  int getValueFor(int argument), gets result value from table
  //  void putValue(int argument, int result), places arg/result into table

  protected pointcut memoizedFunction(int x):
    call(static int MyClass.myFunction(int)) && args(x);

  int around(int arg): memoizedFunction(arg) {
    if(memo.hasValueFor(arg)) {
      return memo.getValueFor(arg);
    } else {
      result = proceed(arg);
      memo.putValue(arg, result);
      return result;
    }
  }
}
```

Reflection at Joinpoints

The AspectJ API provides a reflective form, accessible within advice bodies through the special variable `thisJoinPoint` of type `org.aspectj.lang.JoinPoint` [Tea02a]. This form provides reflective access to the program state at a joinpoint (the point that triggered the advice) as well as static information about the advice:

- the set of arguments (parameters to a method, new value for an assignment, etc.) at the joinpoint
- joinpoint kind (method call, variable read, etc.)
- signature at the joinpoint (the method, field, or type signature), including the declaring type, modifiers, and name.
- location in source code of the joinpoint
- the object currently executing (same as `this()` pointcut)
- the target object of the current code (same as `target()` pointcut)

Example Logging

The primary use of these reflection mechanisms is for tracing and logging applications. The following aspect (from [Tea02a]) provides basic logging facilities for all public methods in any class in any package whose name begins with "`com.bigboxco.`".

```
aspect Logging {
  before(): within(com.bigboxco..*) && execution(public * *(..)) {
    System.err.println("entering: " + thisJoinPoint);
    System.err.println("  w/args: " + thisJoinPoint.getArgs());
    System.err.println("      at: " + thisJoinPoint.getSourceLocation());
  }
}
```

Inter-type member declarations

AspectJ allows aspects to introduce members to other types. These declarations are identical in form to declarations in those types themselves, except that the member's name is prefaced by a type pattern. The type pattern specifies into which types the member will be introduced. Within the body of introduced methods and constructors, `this` refers to the enclosing object, not to the aspect where the member is declared.

For example, the following declaration introduces a `clone()` method to the `Point` class:

```
          public Object Point.clone() { return super.clone(); }
```

Fields may also be introduced to other types in the same way.

Creating inheritance relationships

Using the `declare parents` construct, aspects can declare a superclass and implemented interfaces on classes. The statement

```
  declare parents: C extends D;
```

declares that the superclass of C is D. This is only legal if D is declared to extend the original superclass of C and is not a subclass of C. Interfaces may be introduced using similar syntax, such as

```
  declare parents: C implements I,J;
```

which declares that class C implements interfaces I and J.

Special aspect declarations

In addition to static introduction of members and inheritance relationships, AspectJ allows three special declared forms: compile-time warnings and errors, and wrapping ("softening") of exceptions. Compile-time warnings or errors associate a pointcut with an error or warning condition and message: the compiler will fail or issue a warning if any joinpoint in the given pointcut may be reached. For example,

```
  declare error: call(Singleton.new(..))
      && !withincode(Singleton Singleton.getInstance()): "bad construction";
```

will cause compilation to fail with the message *bad construction* if any calls to any constructor of the class Singleton are found outside of its `getInstance()` method. Warning declarations have the same syntax but use the keyword `warning` instead of `error`.

Additionally, an aspect may specify that a particular kind of exception, if thrown at a joinpoint, should bypass Java's usual static exception checking and be thrown as an `org.aspectj.lang.SoftException`, which is a `RuntimeException` and thus does not need to be declared. For example, the declaration

```
  declare soft: Exception: execution(void main(String[] args));
```

would have a similar effect to the advice

```
  void around() execution(void main(String[] args)) {
    try { proceed(); }
    catch(Exception e) {
      throw new org.aspectj.lang.SoftException(e);
    }
  }
```

except that `declare soft` also affects static exception checking.

Example Cloneable The aspect below (from [Tea02b]), which makes the Point class cloneable, demonstrates static crosscutting in AspectJ.

```
aspect CloneablePoint {
  declare parents: Point implements Cloneable;

  declare soft: CloneNotSupportedException: execution(Object Point.clone());

  public Object Point.clone() { return super.clone(); }
}
```

The aspect CloneablePoint does three things:

1. declares that the `Point` class implements `Cloneable`,
2. declares that the `clone()` method in `Point` should have its checked exceptions converted to unchecked exceptions, and
3. adds a method that overrides the `clone()` method inherited by `Point`.

1.6 Contributions of this Thesis

1.6.1 Updates of Object-Oriented Refactorings

In order to preserve meaning when transforming a program that contains aspect code, it may be necessary to modify references within aspect code that refer to other parts of the program. This is conceptually similar to the reference changes that are required in refactoring a plain OO program but involves some unique challenges. These stem from the great variety of ways that type patterns or pointcuts can refer to program joinpoints; because many different pointcut expressions can select the same set of joinpoints in a given program, it is impossible to predict exactly what references will look like. For example, if `Foo` is the only class in a program that contains a `setX(int)` method, and that method is the only method in the program that takes one integer parameter, the following pointcuts are all equivalent:

```
  call( Foo.setX(int) )
  call( Foo.set*(..) )  // assumes setX is the only set* method in Foo
  call( *.setX(int) )   // assumes no other class has a setX(int) method
  call( *(int) )        // assumes no other class has a method taking an int
  call( *.setX(..) )    // assumes no other class has any method named setX
```

However, these pointcuts would not be equivalent after some simple OO refactorings, such as renaming a method. The first pointcut would become empty if the name of `setX` were changed, and the last pointcut would grow if a method *elsewhere* were renamed to `setX`.

As noted in the comments above, these equivalent pointcuts depend on increasingly unstable assumptions about overall program structure—but each expression is allowed by the language, and legitimate cases for using expressions like these can be made. Therefore, each pointcut or type pattern that may reference elements affected by a refactoring must be interpreted to determine if changes to it are required. Because OO refactorings can both add and remove joinpoints from a particular pointcut expression or type pattern, changes to a pattern or pointcut depend both on the way it is expressed and the refactoring in use.

This thesis describes how pointcuts and type patterns in the AspectJ language can reference joinpoints in the program, and how these references need to be changed in order to preserve behavior when OO refactorings are applied. The OO refactorings considered are the set of about 20 *fundamental* refactorings supplied by Opdyke [Opd92]; many complex refactorings can be composed from these.

1.6.2 New AOP Refactorings

Perhaps the greatest opportunity in combining AOP and refactoring lies in using refactoring to improve a design by employing AOP techniques. This is the most novel and ambitious goal of this thesis: it encapsulates envisioning clear design goals and fashioning particular refactoring techniques in a field with weak consensus on design and little practical experience. Nevertheless, the refactorings are intended to be of more than theoretical interest and an important area of future work will be to test them in the growing AOP software engineering practice.

The distinguishing goal of AOP is to enable the encapsulation of crosscutting concerns. Hence the refactorings introduced in this thesis have as their goal the extraction and encapsulation of crosscutting concerns. In particular, the refactorings are intended to help a programmer improve the structure of crosscutting concerns using AOP techniques. This is a complex aim, but can be reduced into some sub-goals:

- extracting before/around/after advice (for a given pointcut)

  For a pointcut either given explicitly or extrapolated from code selections, it may be desirable to encapsulate behavior occurring before, around, or after all joinpoints in that pointcut. For example, an authorization check may precede all data displays; this could be abstracted into before call advice.

Example Refactoring: Before

```
user.requireSecurity(UFOinfo.securityLevel());
user.send(UFOinfo);
...
user.requireSecurity(budgetDetails.securityLevel());
user.send(budgetDetails);
```

After

```
before(User u, SecureItem s):
  call(User.send(SecureItem)) && target(u) && args(s)
{
  u.requireSecurity(s.securityLevel());
}

user.send(UFOinfo);
...
user.send(budgetDetails);
```

- moving code between aspects and other program components

  Because aspects can introduce methods, fields, and inheritance relationships to classes or interfaces in a program, it is possible to refactor a program by moving these declarations into or out of aspects.

  - extracting disjoint state into static introduction

    A special case of moving code, where some simple analysis can identify disjoint parts of a class such as a set of methods and fields not referenced by other parts of the class. These methods and fields could be moved into a new aspect and their declarations changed so they are statically introduced into the class. One example would be to extract methods associated with a particular interface into an aspect:

    Example Refactoring: Before

```
interface SecureDocument {
  static final int OBVIOUS = 0, TOP_SECRET = 1; ....
  public int getSecretLevel();
}

class WarPlan implements SecureDocument {
  public int getSecretLevel() { return OBVIOUS; }
  public String getData();
  ...
}
```

After

```
interface SecureDocument {
  static final int OBVIOUS = 0, TOP_SECRET = 1; ....
  public int getSecretLevel();
}

aspect WarPlanSecrecy {
  declare parents: WarPlan implements SecureDocument;
  public int WarPlan.getSecretLevel() { return OBVIOUS; }
}

class WarPlan {
  public String getData();
  ...
}
```

– extracting repeated code (across inheritance trees) into static introduction

This is similar in form to the disjoint state extraction, but is based on a different kind of analysis. A set of classes (possibly within a certain inheritance pattern) can be searched for equivalent code declarations, which can be moved into an aspect and declared with static introduction. For example, the ability to add comments to nodes in a program's syntax tree is disjoint from the other program element properties that a node encapsulates.

Example Refactoring: Before

```
class MethodDeclaration {
  MethodBody getBody() { ... }
  String getName() { ... }
  ...

  protected DocComment dc;
  void setDocComment(..) { ... }
  DocComment getDocComment() { ... }
}

class VariableDeclaration {
  Type getType() { ... }
  String getName() { ... }
  ...

  protected DocComment dc;
  void setDocComment(..) { ... }
  DocComment getDocComment() { ... }
}
```

After

```
class MethodDeclaration {
  MethodBody getBody() { ... }
  String getName() { ... }
  ...
}

class VariableDeclaration {
  Type getType() { ... }
  String getName() { ... }
  ...
}

aspect CommentHandling {
  protected interface CommentHandler { }
  declare parents: (MethodDeclaration | VariableDeclaration)
    implements CommentHandler;
  protected DocComment CommentHandler.dc;
  void CommentHandler.setDocComment(..) { ... }
  DocComment CommentHandler.getDocComment() { ... }
}
```

These examples have demonstrated high-level refactorings. These and others are described in Chapter 6.

1.6.3 New Fundamental Refactorings

Based on observations about larger goals in AOP refactoring, this thesis describes requirements for the execution of several AOP refactorings. These refactorings are presented in terms of a new set of fundamental refactorings; the high-level refactorings are expressed in terms of these and the updated fundamental OO refactorings. The new fundamental refactorings are described in detail, including limitations on when they can be applied statically.

Chapter 2

Requirements for Behavior Preservation

Refactorings must preserve the behavior of a program. This chapter describes a set of rules designed to constrain program transformations to preserve program behavior. These constraints fall into three categories: programming language requirements, program properties that must be preserved, and equivalence of semantics. First, the constraints used in OO behavior preservation arguments by Opdyke [Opd92] are reviewed and edited to accommodate a transition from C++ to Java. Then new constraints are defined that will form the basis for behavior preservation arguments in aspect-oriented refactorings.

2.1 Behavior Preservation

Of the many ways we can imagine to transform a program, there are quite a few we might call *behavior preserving.* These vary quite a bit in their formality, complexity, and scope: on one end are changes such as those made by optimizing compilers, like constant folding and dead code elimination. These well-defined transformations apply in specific situations according to strict prerequisites. On the other end are changes made almost exclusively by people, like replacing a certain data structure or algorithm with a more efficient one. The outside program will behave identically as long as the data structure or algorithm works equivalently, but it takes an informal, expert judgment to recognize this equivalence.

The refactorings presented in this thesis fall closer to compile-time optimizations because they are presented formally based on decidable prerequisites. Refactorings, however, are designed with a broader goal: to assist the programmer in achieving all kinds of design improvements. Many desired design improvements—such as substituting an algorithm—might not be achievable purely by iterative application of these refactorings. Still, known refactorings can often assist in much of the process. The refactoring descriptions presented here have the following characteristics:

1. **Each refactoring describes a kind of change that is generally useful in the development of object-oriented and aspect-oriented software.** The changes a refactoring entails are explained in terms of common, straightforward program alterations *or* more complex ones that are defined in this thesis.

2. **The preconditions of each refactoring are decidable by static analysis, at least conservatively** (i.e., to the point of avoiding false positives). For example, suppose some class X contains a method `foo()` and we want to know if an overriding method `foo()` in a subclass Y of X is ever called. Even if no variable or expression that has static type Y has the method `foo()` called on it, it is possible that an object of type Y is substituted at runtime and the method is called on that—we can't always know for sure because of dynamic dispatch. But often we can find that `foo()` is never called on any value that could possibly have type Y, and infer the precondition based on that.

3. **Each refactoring is argued to be behavior preserving by showing that it satisfies certain conservative constraints on program transformations that imply behavior preservation.** Establishing constraints that are flexible enough to permit interesting changes, but simple enough to actually satisfy, is the primary goal of this chapter.

2.1.1 Assumptions and Limitations

Several important assumptions are made in the statements of preconditions, transformations, and behavior preservation arguments presented later in this thesis. The first assumption is that any program we wish to refactor is syntactically and semantically valid—it compiles. This assumption simplifies the preconditions each refactoring requires. Though a particular error that causes a program to fail compilation may not be pertinent to a given refactoring, questions of when and how these interdependencies can be determined is outside the scope of this thesis.

Another assumption that is required to properly execute many of the analyses and transformations is that we have access to the entire program. That is, we assume we can read and modify all the relevant code, and we have no responsibility to preserve the behavior of anything besides our program's `main` method[1]. In a large application we might need to analyze and alter hundreds of classes if a commonly used class's name is changed, so we assume access to all of these.

This assumption is most limiting when refactoring a library module. Because a library is committed to supporting a certain interface, refactorings that break that interface ought not be allowed, because this would risk changing the behavior of some client code. To avoid this challenge, we assume that there is no client code outside our view.

[1] Though multiple classes involved in a program may have `main` methods, we assume in this thesis that a program's behavior is determined by a single invoked main method.

Finally, a significant limitation on where the refactorings described in this thesis are valid is imposed by reflection. Like other program constructs, uses of reflection may form dependencies on structures that our refactorings are designed to change. However, because these dependencies are expressed in dynamic values rather than in static code, the details of these dependencies are hard or impossible to determine statically. For instance, if a specific error handler class is specified in a program's configuration file and dynamically instantiated based on its name, renaming that class would cause the program to fail when attempting to load the class under its old name.

2.2 Behavior Preservation Constraints for Refactoring

Our first assumption about a program we want to refactor is that it compiles. A program that does not compile can hardly be thought equivalent to one that does. Thus we reach a first constraint on refactoring:

- In executing a refactoring, we must not transform the program into a state that is forbidden by the programming language.

We might approach the task of satisfying this constraint by being careful about what specific changes are allowed. While many such pitfalls are easy to avoid at each step of refactoring, others—such as subtype cycles—might require explicit checks. (Specific constraints that address these issues are described in section 2.2.1.)

A change that preserves linguistic validity, however, is not always a refactoring. There are in fact many such transformations that *do* change the behavior of a program—without them software development would be impossible! But even when we limit our transformations to those in the spirit of refactoring, there may be errors a compiler will not catch. For example, consider a renaming of the method `f2()` to `f1()` in a class `B`, shown in Figure 2.1. Assuming `A.f1()` and the original `B.f2()` behave differently, this 'refac-

Erroneous Refactoring: Before

```
class A {
  void f1(int x) { ... }
}

class B extends A {
  void f2(int x) { ... }
  void main(int x) { f1(x) ... }
}
```

After

```
class A {
  void f1(int x) { ... }
}

class B extends A {
  void f1(int x) { ... }
  void main(int x) { f1(x) ... }
}
```

Figure 2.1: Renaming a method in class `B`: `f2()` $\longrightarrow$ `f1()`

toring' has failed to preserve meaning because the new name, `f1`, overrides a declaration previously used by the `main()` method. One rule that would have prevented the incorrect application of this refactoring is that *a refactoring should not cause previously* *visible inherited members to be overridden.* The following sections describe more of these rules, which will form the basis of our behavior-preservation arguments.

2.2.1 Language Requirements

The following constraints are programming language rules commonly enforced by a compiler. Since any refactoring must transform a valid (compiling) program into another valid program, a refactoring cannot violate these properties. While there are many language constraints a refactoring must observe, the ones listed below typically require explicit checks before refactoring because they are easily violated by improper application of basic refactoring steps such as changing types and renaming or moving program elements.

L1. Each class or aspect must have one direct superclass or superaspect that is not its direct or indirect subclass or subaspect.

L2. Each type (class, aspect, or interface) must have a unique name.

L3. Each variable must have a unique name in its scope.

L4. Each method in a type must have a unique signature.

L5. The program must be type safe.

L6. Inherited fields may not be overridden.

L7. Rules for `extends` and `implements`:

   (a) An interface can only extend other interfaces. An interface cannot implement another interface.

   (b) A class can only extend another class.

   (c) An aspect can only extend another aspect or a class.

   (d) A class or an aspect can only implement interfaces.

L8. If two methods overload a certain method signature, one's signature must be (unambiguously) more precise than the other's.

2.2.2 Preserving Inheritance Properties

The following constraints require that as a program is refactored, certain inheritance relationships and requirements are maintained. These are conservative constraints; there are ways we could violate them in the course of a legitimate refactoring. But for our purposes, they provides a good compromise, allowing us to incorporate a variety of useful changes into our refactorings while limiting the potentially complex consequences of a refactoring throughout an inheritance hierarchy.

I1. If a method that is inherited (including abstract methods in interfaces) is changed, those changes to it must be compensated for in subtypes.

An inherited method can only be overridden when the overriding method's signature is compatible with its own. We require that a refactoring preserve these kinds of overriding relationships. Thus, when applying a refactoring that changes a method's signature, it is necessary to correspondingly change the signatures of methods in subtypes of the containing type.

(A similar constraint would apply to fields, except that the Java language does not permit overriding of inherited fields.)

I2. A class or aspect that implements any interfaces or abstract classes should continue to satisfy the requirements imposed by its interfaces or abstract superclasses after refactoring.

2.2.3 Semantic Equivalence

Each refactoring alters a part of the program but preserves the way that part behaves, at runtime, with respect to some external interface. For example, consider applying a refactoring that inlines a method call within an existing method. The existing method has the same signature and computes the same values as before the refactoring, so that method's behavior is preserved. From this we can derive an intuitive model for whole-program behavior preservation: we simply apply this requirement to a program's `main` method. Thus, if we consider our program as purely a function, then any change that does not change the `main` function's mapping from input to output values is a refactoring.

Unfortunately, this model can only offer an intuitive explanation for refactoring in general. Because the question of whether a change will alter the function that a program computes is undecidable, the question of whether a given change is behavior-preserving is also undecidable. However, specific refactorings can be argued to be behavior-preserving in terms specific to their task. From these fundamental refactorings we can assemble more sophisticated high-level refactorings.

Opdyke [Opd92] offers the metaphor of a circle delimiting the domain of each refactoring. There is some circle we can draw around part or all of the program, within which we can make changes *provided that* at the end, things outside that circle can interact with it in the same ways and to the same ends as they did before the changes. The circle's size and location vary depending on the refactoring; the circle may include one method, as above, or much of a program, such as if a frequently referenced variable is renamed.

This constraint permits several useful changes:

1. expressions can be simplified

   Since an expression is used to compute a value, an equal expression (such as a simplification) can replace it without changing program behavior.

2. dead code can be removed

Because unreachable code does not have any effect on a program's execution, it can be removed without changing behavior.

3. conditionals can be simplified based on invariant conditions

   If we know that a certain condition is guaranteed to hold at every evaluation of a conditional expression, that expression can be simplified based on the invariant condition.

4. unreferenced variables, methods, and classes can be added or removed

   As with dead code, variables, methods, and classes that are never referenced in a program's execution do not affect behavior. Note that this allows several chunks of code that reference each other to be removed all at once, as long as none of them are referenced from the parts of the program that actually run (i.e., are reachable from the program's `main` method, which is always assumed to be referenced).

5. a variable's type can be changed, as long as each operation referenced on the variable is defined equivalently for its new type, and all assignments involving that variable remain type safe

   For example, depending on the interface used on a given variable, it may be possible to change its declaration to use a subtype or supertype. If the set of operations invoked on the variable are all inherited from the original type in a subtype, the declaration may be changed to use that subtype instead. Similarly, if all of the invoked operations are inherited from the supertype, we can generalize to that supertype without changing program behavior.

   Other changes may be possible in some cases, such as when one type delegates to another for a set of operations. But the cases of type substitutability that we can detect automatically are limited.

6. references to a field or method defined in one class can be replaced with references to other fields or methods that are equivalently defined

   If two variables are known to refer to the same object (decidable in limited cases), or if two methods have equivalent bodies (equivalent code and variable references), references to one can be replaced with references to the other.

Many of these changes are commonly made automatically by optimizing compilers. While a compiler optimization, like a refactoring, is a static program transformation, its motivation is entirely different. In fact, for every change allowed by the constraint of semantic equivalence, its inverse must also be allowed. For example, an important refactoring is *create empty class.* This is a useful step in evolving a design, but considered in isolation is just new dead code—a change an optimizing compiler ought never introduce.

2.3 Properties of AOP behavior preservation

With aspect-oriented constructs, program elements declared outside a class can affect its structure and execution. Thus when aspects are present in a program, our refactorings must preserve these AOP semantics in ways analogous to the preserved OO semantics.

2.3.1 Language Requirements

The first consequence of allowing AOP constructs is that our language requirements are extended. The three main AOP features—inter-type member declaration, declaration of new inheritance relationships, and advice—each have several effects on language requirements.

Inter-type member declarations, which can function exactly like local declarations in a target type, can be declared in any aspect. This affects the requirements that members of a class have unique names and signatures, as both introduced and local declarations need to be checked for conflicts.

Aspects can also declare new inheritance relationships by declaring a class to implement an interface or by assigning it a new superclass. (The new superclass must be a subclass of the original superclass.) These declarations must be taken into account when changing inheritance relationships in a refactoring, as the first language requirement (that each class must have one direct superclass that is not its subclass) may be violated. For example, a class's supertype could be changed in a way that conflicts with an introduced supertype.

2.3.2 Crosscutting Structure and Semantic Equivalence

A primary new feature of aspect-oriented programs is crosscutting structure. This poses a special challenge in refactoring, because to preserve semantic equivalence, crosscutting structure must apply at semantically equivalent points before and after refactoring. AspectJ programs implement crosscutting structure using two mechanisms: inter-type declarations and advice. While these mechanisms differ substantially, they do share some common ways of referring to other parts of a program.

Inter-type declarations, which include introduction and the `declare parents`, `warning`, `error`, and `soft` constructs, are features that alter a program's static structure based on target types that are either explicitly named or that match a given type pattern. Therefore, in carrying out a refactoring we must ensure that the targets and effects of inter-type declarations after refactoring are semantically equivalent to their targets and effects before refactoring. Additionally, introduced methods contain code bodies. This code, if it is itself reachable, can refer to program parts that might not be referenced elsewhere; thus it must be considered in analyses checking for dead or unreferenced code.

Advice is based on pointcuts, which are sets of joinpoints—well-defined events in a program's execution. Some pointcuts, such as `call`, `get`, and `set`, are statically determinable: given full source code, we can identify exactly where all appropriate method calls and field accesses occur. This can be used to efficiently instrument advice when aspects are woven into classes statically. Other pointcuts, however, cannot generally be determined statically. These include `cflow`, `cflowbelow`, `if`, and some cases of `this`, `target`, and `args`. In a refactoring we'd like to make sure each pointcut is left with either the same joinpoints or semantically equivalent joinpoints to those it contained before the refactoring. None should be added and none removed. In some cases, we will argue that a refactoring is meaning-preserving with respect to advice by showing that the meanings of pointcuts are preserved.

Another possibility, however, is to present the argument in terms of the patterns that parameterize the pointcut itself: if the patterns match semantically equivalent program elements, then the pointcut will match semantically equivalent joinpoints. This is a stronger requirement than simply that the pointcut contain an equivalent set of joinpoints, because while individual patterns may each match many elements, they could parameterize a pointcut that is small or empty. For instance, the pointcut

```
call( public int Foo.getLength() )
```

implies the following dependencies at the pattern level:

- a method called `getLength`,
- with no arguments,
- in a class called `Foo`,
- returning an `int`, and
- declared to be `public`.

But unless this method exists and there actually *are* calls to it, this `call` pointcut will be empty. But while pattern equivalence is a stricter condition, it offers some advantages. The exact elements that match a pattern can always be determined statically (given complete source code), and indeed quite simply. Thus, when possible, behavior preservation arguments are based on preserving the meaning of patterns. Details on analyzing and manipulating patterns are presented in Chapter 4, Working with Patterns.

Finally, advice bodies also contain code. This code, if it is itself reachable, can refer to program parts that might not be referenced elsewhere; thus it must be considered in analyses checking for dead or unreferenced code.

Chapter 3

Low-Level Refactorings Reconsidered

Low-level refactorings are small refactorings useful both on their own and in constructing more complex refactorings. This chapter reviews the low-level object-oriented refactorings presented by Opdyke [Opd92], each edited for Java (from the original C++) and augmented with extra preconditions and steps in order to guarantee behavior preservation in AspectJ.

3.1 Creating a Program Entity

3.1.1 Create Empty Class

Define a new class with no locally defined members, optionally with a designated superclass.

Preconditions:

1. The class's name does not conflict with an already existing type (class, interface, or aspect).

2. ✂[1] The class must not be subject to a combination of inter-type parent declarations that will cause it to fail compilation. The possible conditions are:

   (a) an illegal combination of superclasses

   (b) inappropriate kinds of supertypes (e.g. if the class would be declared to extend an interface)

[1]The cutting scissors symbol (✂) denotes a precondition as particularly concerned with AOP behavior preservation constraints.

  (c) superinterfaces or abstract superclasses that an empty class cannot actually implement

3. ✂ The class must not be subject to a set of inter-type member declarations that will cause it to fail compilation (e.g. if two fields with the same name are introduced).[2]

These three preconditions guarantee that the refactored program will compile. Precondition 1 guarantees a unique name (constraint L2). Precondition 2 ensures a unique superclass (L1) and legal `extendsimplements` declarations (L7). Precondition 3 guarantees that members introduced to the class do not cause compilation to fail, whether due to local conflicts (L3, L4), redefinition of inherited fields (L6), or illegal overloading (L8). Semantically equivalent behavior is guaranteed because a new class is never instantiated or referenced; thus the program's behavior does not change when it is added.

3.1.2 Create Interface

Define a new interface with no locally defined members, optionally with any number of designated superinterfaces.[3]

Preconditions:

1. The interface's name does not conflict with an already existing type.

2. ✂ The interface must not be subject to a combination of inter-type parent or member declarations that will cause it to fail compilation.

These preconditions guarantee that the new interface will compile: the new interface will have a unique name (L2), will follow rules about `extends` and `implements` declarations (L7), and will not include illegal members (L3, L4, L6, L8). The only check necessary to verify that the combination of superinterfaces is valid is to verify that any required overloadings are valid (L8).

As with a class, because the interface is never referenced, the behavior of the program does not change when it is added.

3.1.3 Create Field

Add an unreferenced field to a class.

Preconditions:

1. The field's name does not conflict with an existing locally defined, inherited, or introduced field.

[2]An introduced member would be illegal on a class c if it fails to compile in c. In the case of a newly created class, a member introduced to it might e.g. refer to another member that does not actually exist, causing compilation to fail; or two fields with the same name could be introduced.

[3]Because C++ does not include interfaces, this refactoring is not based on one in [Opd92], but is an obvious analog.

2. If the field is not declared with private visibility, no name conflict occurs in any subclasses of the target class.

These two preconditions guarantee that no name conflicts are created in the class or its subclasses (L3, L6). Behavior is not changed because the field is not referenced, and because pointcuts can only pick out uses (assignments and reads) of a field, not its declaration. (Note that this new field does not include an initial assignment, aside from Java's implicit default initializer.)

3.1.4 Create Method

Add a locally defined method to a class. The method is either unreferenced or identical (in signature and body) to an already inherited method.

Preconditions:

1. The new method will compile as a member of the target class.
2. If the new method will overload an existing method (either in the target class or in its subclasses), it must either be more general (thus not capturing any calls that would have invoked the existing method) or more precise and semantically equivalent to the method it overloads.
3. If the target class has an inherited method that will be overridden by the new method, either that method is unreferenced on the target class and its subclasses, or the new method is semantically equivalent (e.g. identical) to the method it overrides[4].

The first precondition implicitly guarantees no signature conflicts with current locally declared or introduced members of the class (L4). The second precondition guarantees valid overloading (L8). The final precondition guarantees that even if the new method overrides an inherited method, program behavior is preserved.

3.2 Deleting a Program Entity

3.2.1 Delete Unreferenced Class

Preconditions:

1. The target class is never referenced.

If a class is unreferenced, it will not be loaded or initialized and none of its members will be called. Hence removing it does not affect any pointcuts in the program. Even if patterns refer to the class or its members specifically, no possible joinpoints will ever be reached. All program properties are preserved.

[4]As mentioned in the previous chapter, it is not always possible to determine whether a particular method in a given class is referenced because of dynamic method dispatch. In many cases, however, it is possible to determine conservatively that a method is not referenced.

(This is still a strict definition of *referenced* as referenced directly in program statements, and does not include matching a given type pattern. For example, suppose the class is a target of static inter-type declaration. Once the class is removed, the type pattern that matched it will still cause the introduction to target any other types it originally matched. If the pattern fails to match anything, this does not signal an error in AspectJ.)

3.2.2 Delete Unreferenced Field

Preconditions:

1. The field is never referenced (including implicitly in an initializer).

By similar rationale to deleting a class, deleting an unreferenced field does not affect program behavior.

3.2.3 Delete Set of Unreferenced Methods

Delete a set of methods from a class.

Preconditions:

1. Each method to be deleted is
   - not referenced from outside the set of methods to be deleted,
   - *or* redundant because a semantically equivalent method is inherited[5],
   - *or* redundant because it overloads a semantically equivalent method with a more general signature.

This refactoring is limited to unreachable methods. (Each program has a main method that is always implicitly referenced.) Like an unreferenced field or class, an unreferenced method does not trigger any pointcuts. If a semantically equivalent method is inherited, removing a locally defined method simply causes that inherited method to be invoked instead. If a semantically equivalent method is overloaded and we remove the more specific method, the more general method will simply be invoked in its place. Thus, behavior is preserved.

3.3 Changing a Program Entity

3.3.1 Rename a Class

Change the name of a class, including references throughout the program.

Preconditions:

[5] There cannot be a redundancy because a method semantically equivalent to a method we want to remove is introduced to a class from an aspect. Such an introduction, where the signature of an introduced method conflicts with that of a locally declared method, is an error in AspectJ.

1. The new name doesn't conflict with an already existing type.

To keep this change behavior preserving in aspect-oriented programs, it may be necessary to modify type patterns. The following conditions must hold:

1. ✄ If a type pattern matched this class before renaming and does *not* match afterwards, it should be extended to match.
2. ✄ If a type pattern did not match this class before renaming and *does* match afterwards, it should be narrowed to avoid matching.

The precondition ensures distinct type names (L2). The new steps guarantee that, because the meanings of patterns are changed appropriately, references are semantically equivalent. Thus, behavior is preserved.

3.3.2 Rename a Variable

Change the name of a variable. The name change is reflected throughout its scope. (The scope includes subclasses if the variable is an inherited field.)

Preconditions:

1. The new name doesn't conflict with an already existing variable in the same scope (or in subclasses if applicable).

If the variable being changed is a field, the following conditions must also hold:

1. ✄ The new name doesn't conflict with a field introduced into the same type (or its subtypes) from an aspect.
2. ✄ If a field pattern matched this field before renaming and does *not* match afterwards, it should be extended to match.
3. ✄ If a field pattern did not match this field before renaming and *does* match afterwards, it should be narrowed to avoid matching.

The precondition ensures that the new name doesn't conflict with other variables (L3). The new steps guarantee that, because the meanings of patterns are changed appropriately, references are semantically equivalent. Thus, behavior is preserved.

3.3.3 Rename a Method

Change the name of a method m in a class c, and any overriding methods defined in subclasses, as well as all references.

Preconditions:

1. If new signature does not conflict with, overload, or override an already existing local, introduced, or inherited method in c.

2. If m is not declared `private`, no subclass of c that inherits m defines a method with the new name and a signature that would overload or override m.

The following conditions must also hold:

1. ✃ If a method pattern matched m before renaming and does *not* match afterwards, it should be extended to match.

2. ✃ If a method pattern did not match m before renaming and *does* match afterwards, it should be narrowed to avoid matching.

The preconditions ensure that the new signature doesn't conflict with other methods (L4) and preserves overriding and overloading relationships that are legal (L8). The new steps guarantee that, because the meanings of patterns are changed appropriately, patterns are semantically equivalent. Thus, behavior is preserved.

3.3.4 Change Type

Change the type of a set V of variables/parameters and a set M methods to a new type t. (Change the types of the variables and the return types of the methods.)

Preconditions:

1. Each expression and assignment involving a variable in V or method in M would remain semantically equivalent and type safe if its type were changed to t.

The following conditions must also hold for each variable $v \in V$ that is a field:

1. ✃ If a field pattern matched v before changing its type and does *not* match afterwards, it should be extended to match.

2. ✃ If a field pattern did not match v before changing its type and *does* match afterwards, it should be narrowed to avoid matching.

And for each method $m \in M$:

1. ✃ If a method pattern matched m before changing its type and does *not* match afterwards, it should be extended to match.

2. ✃ If a method pattern did not match m before changing its type and *does* match afterwards, it should be narrowed to avoid matching.

The precondition ensures that type safety is preserved (L5). The new steps guarantee semantically equivalent references from patterns. Thus, behavior is preserved.

3.3.5 Change Access Control Mode

Change the access control (visibility) mode of a member (method or field) i in a class c from *oldMode* to *newMode*. The possible modes are `private` (visible in containing class only), `protected` (in package and any subclasses), `public` (visible wherever the container is visible), and default (package visibility).

Preconditions:

1. The class c will compile with the new access control mode on i.
2. If *newMode* is `private`:
    - (a) The member is not referenced outside the class where it is defined.
    - (b) The member does not override an inherited member.
3. If *newMode* is default:
    - (a) The member i is not referenced outside the package where it is defined.
    - (b) The member does not override an inherited member that has non-default visibility.
    - (c) If the member is a field, no subclass of c in the same package as c declares a field with the same name.
    - (d) If i is a method
        - i. it is not overridden with `private` visibility in a subclass, and
        - ii. if *oldMode* is `private`, no subclass of c in the same package as c declares a method that would be overloaded more specifically by i.
4. If *newMode* is `protected`:
    - (a) The member is only referenced from the package where it is defined or from subclasses of c.
    - (b) The member does not override an inherited member that has `public` visibility.
    - (c) If the member is a field, no subclass of c declares a field with the same name.
    - (d) If the member is a method,
        - i. it is not overridden with `private` or default visibility in a subclass, and
        - ii. if *oldMode* is `private` or default, no subclass of c in the same package as c declares a method that would be overloaded more specifically by i.
5. If *newMode* is `public`:
    - (a) If the member is a field, no subclass of c declares a field with the same name.
    - (b) If the member is a method,
        - i. it is not overridden with non-`public` visibility in a subclass, and

ii. if *oldMode* is `private` or default, no subclass of c in the same package as c declares a method that would be overloaded more specifically by i.

The following conditions must also hold:

1. ✄ If a field or method pattern matched i before changing its access control mode and does *not* match afterwards, it should be extended to match.

2. ✄ If a field or method pattern did not match i before changing its access control mode and *does* match afterwards, it should be narrowed to avoid matching.

The first precondition ensures that requirements imposed on c by abstract superclasses or superinterfaces are still met. The preconditions that require current references to i to remain legal after the refactoring ensures both that the program will still compile and that the same members are referenced (these could change in the case of an overloaded method). The new steps guarantee semantically equivalent references from patterns. Thus, behavior is preserved.

3.3.6 Add Parameter to Method

Add a new parameter of type t to the declaration of a method m in a class c, and to methods that override it in subclasses. In each call to m, add an argument *default* (an expression that computes a value of type t; possibly `null`).

Preconditions:

1. The name of the new argument doesn't conflict with another variable or parameter in the method's scope. This property must hold for all methods that override m in subclasses.

2. A method with the same signature as m *before* adding a parameter is not inherited from a superclass or implemented interface.

3. If m will override an inherited method *after* adding a parameter, the inherited method is unreferenced on c and its subclasses, or m is semantically equivalent to it.

4. The method m with a parameter added does not create an overloading of a method already in c or in any subclass of c.

5. *default* is visible in all places where m is called.

The following must also hold for m:

1. ✄ If a method pattern matched m before adding a parameter and does *not* match afterwards, it should be extended to match.

2. ✄ If a method pattern did not match m before adding a parameter and *does* match afterwards, it should be narrowed to avoid matching.

The first precondition guarantees no name collisions. The second precondition guarantees that inheritance properties (I1, I2) are preserved. The third precondition guarantees that overriding will not change behavior. The fourth precondition guarantees that no overloading occurs that might cause calls to m to be captured by another method (or vice-versa). Because the new parameter is unreferenced, and with the new steps to ensure semantic equivalence of patterns, behavior is preserved.

3.3.7 Remove Unreferenced Method Parameter

Delete a parameter p from a method declaration m in a class c. Unless m is declared with `private` visibility, also delete this parameter from all subclasses where an overriding method is defined.

Preconditions:

1. The parameter is unreferenced.
2. The corresponding parameter in each method overriding m is unreferenced.
3. The method m is not itself overriding an inherited method.
4. If m will override an inherited method once the parameter p is removed, the inherited method is unreferenced on c and its subclasses, or m is semantically equivalent to it.
5. After removing p, the method m must not conflict with a locally defined or introduced method declaration in c (and in all subclasses if m is inherited).
6. After removing p, m does not overload a method already in c or in any subclass of c.
7. The expressions passed through p on calls of m have no side effects.

The following constraints must also hold:

1. ✂ If a method pattern matched m before removing a parameter and does *not* match afterwards, it should be extended to match.
2. ✂ If a method pattern did not match m before removing a parameter and *does* match afterwards, it should be narrowed to avoid matching.

By the first two preconditions, p is unreferenced. By the third precondition, inheritance relationships will be preserved. The fourth precondition ensures that a new overriding will not affect behavior. The fifth precondition guarantees unique member names (L4). The sixth precondition guarantees that no overloading occurs that might cause calls to m to be captured by another method (or vice-versa). The last precondition ensures that removing the computation of the parameter at call sites does not affect program behavior. The extra steps ensure meaning preservation of patterns.

3.3.8 Reorder Method Parameters

Reorder the parameters of a method m in a class c. Also reorder parameters analogously in all overriding methods in subclasses, and the actual parameters in all calls to these methods.

Preconditions:

1. The expressions assigned to parameters in calls of m have no side effects. (If two expressions of m do have side effects that are not independent, changing the order in which they are evaluated could change program behavior.)

2. Before refactoring, m does not override a method defined in a superclass of c.

3. If m will override an inherited method once the parameters are reordered, the inherited method is unreferenced on c and its subclasses, or m is semantically equivalent to it.

4. After the reordering, m does not overload a method already in c or in any subclass of c.

The following constraints must also hold:

1. ✂ If a method pattern matched m before reordering parameters and does *not* match afterwards, it should be extended to match.

2. ✂ If a method pattern did not match m before reordering parameters and *does* match afterwards, it should be narrowed to avoid matching.

The first precondition guarantees that the parameters can indeed be reordered without changing program behavior. The second and third preconditions preserve inheritance relationships. The fourth precondition ensures that no overloading occurs that might cause calls to m to be captured by another method (or vice-versa). The extra steps ensure meaning preservation of patterns.

3.3.9 Add Method Body (Concretize Abstract Method)

Add a method body to an existing abstract method with no body.

Preconditions:

1. The method (with its new body) compiles.

Because an abstract method is never called, adding a method body cannot change program behavior, and no new joinpoints pertaining to that method can be reached. Program behavior is preserved.

3.3.10 Remove Method Body (Make Concrete Method Abstract)

Delete a method body from an existing method, making it abstract.

Preconditions:

1. The method is never called.

Because the method is never called, removing its body cannot change program behavior, and no joinpoints pertaining to that method become unreachable.

3.3.11 Replace Field References With Accessor/Modifier Calls

Convert all references to a variable v, except those in its accessor method, to calls to its accessor method. Convert all assignments to v, except those in its modifier method, to calls to its modifier method.

Preconditions:

1. The accessor and modifier methods for v are already defined (to simply get and set v, respectively) on the class that contains v, have the same visibility as v, and are not overridden in any subclasses.

2. ✂ If v is matched by any field patterns used to parameterize `get` or `set` pointcuts, these pointcuts are not intersected with a `within` or `withincode` pointcut.

   - The `within` and `withincode` pointcuts match joinpoints inside methods or classes matching a certain pattern, and are commonly combined with other pointcuts by intersection. For example:

     ```
     set( Foo.a ) && !within( org.foo.* )
     ```

     This pointcut expression matches all sets to the field `Foo.a` that occur in classes outside of packages whose names begin with "`org.foo.`".

     Because the point of this refactoring is to make the assignment happen somewhere else, if a pointcut depends on *where* the assignment occurs, its content will change when this refactoring is applied. This precondition precludes this possibility.[6]

3.3.12 Replace Statement List with Method Call

Replace a statement list L in a method m with the method call mc.

Preconditions:

[6] This precondition can be computed statically using a kind of data flow analysis based on pointcut expressions. Alternatively, a stricter precondition can be adopted. The stricter precondition would require that no `within` or `withincode` pointcut is parameterized with a type or method pattern that matches any class or method where v is referenced. If this is the case, the intersection of any relevant `get` or `set` pointcut with any `within` or `withincode` pointcut that is actually in the program will be empty.

1. The called function, *mc*, is visible from the calling function, *m*.
2. The call to *mc* is semantically equivalent to *L*, i.e. their abstract syntax trees are the same, up to variable renaming, and they reference semantically equivalent items outside their scopes.
3. ✂ The method *mc* is not matched by any method patterns. (This precludes new invocation of advice parameterized by `call`, `execution`, and `withincode` pointcuts.)
4. ✂ If any statement in *L* includes an assignment to or read from a field matched by a field pattern used in a `set` or `get` pointcut, that pointcut is not intersected with a `within` or `withincode` pointcut.
5. ✂ If any statement in *L* includes a method or constructor call matched by a method or constructor pattern used in a `call` or `execution` pointcut, that pointcut is not intersected with a `within` or `withincode` pointcut.

The first two preconditions ensure that the called method behaves identically to the statements it replaces. The rest of the preconditions are AspectJ-specific, and guarantee that this refactoring does not change which points in program execution cause advice to be invoked. To preclude invoking advice on *mc* itself, we require that it is not matched by any method patterns. Further, we do not want our use of the method call to cause invocation of any advice not already invoked in our statement list. Similarly, we do not want our use of the method call to preclude invocation of advice that *was* invoked by a statement in the list. The only way, aside from the dynamically-determinable `cflow` and `cflowbelow` pointcuts, to achieve this kind of limitation in a pointcut is to use `within` or `withincode` pointcuts. By intersecting (&&) these with other pointcuts, it is possible to constrain advice executions to joinpoints occurring in a specific set of types or methods. Rather than attempting to fix these pointcuts to include or exclude specific joinpoints, we simply require as a precondition that they do not apply.

3.3.13 Inline Method Call

Replace a method call *mc* with the body of the called method *m*.

Preconditions:

1. All variables and methods referenced from the body of *m* are visible from the caller that contains *mc*.
2. ✂ The method *m* is not matched by any method patterns. (This precludes losing invocation of advice parameterized by `call`, `execution`, and `withincode` pointcuts.)
3. ✂ If any statement in the body of *m* includes an assignment to or read from a field matched by a field pattern used in a `set` or `get` pointcut, that pointcut is not intersected with a `within` or `withincode` pointcut.

4. ✄ If any statement in the body of m includes a method or constructor call matched by a method or constructor pattern used in a `call` or `execution` pointcut, that pointcut is not intersected with a `within` or `withincode` pointcut.

The first precondition guarantees that the inlining will be valid. The rest of the preconditions are AspectJ-specific, and guarantee that this refactoring does not change which points in program execution cause advice to be invoked. So that the inlining does not cause advice to not be run, we require that the method is not matched by any method patterns.

Further, we do not want our inlining of the method body to cause invocation of any advice not already invoked in our statement list. Similarly, we do not want the inlining to preclude invocation of advice that *was* invoked by a statement in the list. The only way, aside from the dynamically-determinable `cflow` and `cflowbelow` pointcuts, to achieve this kind of limitation in a pointcut is to use `within` or `withincode` pointcuts. By intersecting (&&) these with other pointcuts, it is possible to constrain advice executions to joinpoints occurring in a specific set of types or methods. Rather than attempting to fix these pointcuts to include or exclude specific joinpoints, we simply require as a precondition that they do not apply.

3.3.14 Change a Class's Superclass

Change the superclass of a class c from class s_0 to s'.

Preconditions:

1. Each expression and assignment involving a variable of type c (or any subtype) would remain type safe if the superclass is changed.

2. All members inherited by c from s_0 that are referenced on c or any subclass of c will be replaced by semantically equivalent inherited members from s'. Here, semantic equivalence refers not only to member definition, but pattern matches. That is, if a member m is inherited from s_0 and is to be replaced by m' from s', then:

   - **If m is a method,** m' has an identical name and signature, and all method patterns that match m also match m'.
   - **If m is a field,** m' has the same name and type, and all field patterns that match m also match m'.

3. No newly inherited method from s' creates a more specific overloading of a method in c.

4. No newly inherited method from s' creates an illegal overloading of a method in c.

The following constraints must also hold:

1. ✄ If a type pattern matched c before changing its supertype and does *not* match afterwards, it should be extended to match.

2. ✂ If a type pattern did not match c before changing its supertype and *does* match afterwards, it should be narrowed to avoid matching.

After verifying the preconditions, this refactoring changes the superclass of c to s'. Then it deletes from c any member variables whose semantic equivalents are inherited from the new superclass s'. If such a member variable is introduced to c from an aspect using static introduction, the type pattern for that introduction is narrowed to not include c.

The first precondition guarantees type safety (L5). Precondition 2 ensures that inheritance has the same semantic effects on c. Preconditions 3 and 4 ensure that if overloading occurs, it is valid and does not capture calls to a method in c.

The extra step of fixing type patterns guarantees meaning preservation of patterns. The step of deleting member variables from c that are inherited from s' guarantees that no (illegal) field overridings are created (L6).

3.3.15 Implement an Empty Interface

Any class can be declared to implement an interface that does not contain any members. Program behavior is unchanged.

(This trivial refactoring is used in Chapter 6 within a high-level AOP-specific refactoring.)

3.4 Moving a Field

3.4.1 Move Field to Superclass

Move a field x to class c from all subclasses of c where it is defined. If x has `public` visibility, its visibility in c will also be `public`; otherwise, the visibility will be `protected`.

Preconditions:

1. The field declarations that have the same name as x are identical across all subclasses that declare such a field.

2. The visibility of these declarations is either `public` or it is `protected` and all references to the field occur in subclasses of c.

3. The superclass c does not already contain a (`private`) field with the same name as x.

The following conditions must also hold:

1. ✂ If a field pattern matched x in a subclass before and does *not* match x in the superclass, it should be extended to match.

2. ✂ If a field pattern did not match x in a subclass before and *does* match x in the superclass, it should be narrowed to avoid matching.

The first precondition guarantees that the field can be safely pulled up into the superclass (including avoiding name conflicts , L3). The second precondition restricts the allowed visibility modes in order to avoid overloading and overriding concerns encountered when increasing a field's visibility. (This may be separately addressed by the Change Access Control Mode refactoring, §3.3.5.)

The new steps ensure semantic equivalence of patterns. Thus, behavior is preserved.

3.4.2 Move Field to Subclasses

Move a field x from its current containing class c to each of c's direct subclasses.

Preconditions:

1. The field x is not referenced from within c.
2. The visibility of x is `protected` or `public`, or its visibility is default and all subclasses of c are in the same package as c.
3. The type of x is visible in all subclasses.

The following conditions must also hold:

1. ✂ If a field pattern matched x in the superclass before and does *not* match each x in the subclasses, it should be extended to match.
2. ✂ If a field pattern did not match x in the superclass before and *does* match x in any subclass, it should be narrowed to avoid matching.

The preconditions guarantee that the field can be safely moved down into the subclasses, where it would already be inherited. The new steps ensure semantic equivalence of patterns. Thus, behavior is preserved.

Chapter 4

Working with Patterns

Patterns are the most basic mechanisms used in AspectJ to describe the targets of crosscutting structure, and often need to be modified to preserve meaning when refactoring. Pointcuts and inter-type declarations in AspectJ can be parameterized with patterns that pick out types, fields, methods, and constructors. This chapter discusses some ways that these patterns can be analyzed and modified for use in refactorings.

4.1 Modifying Patterns

Many of the refactorings described in Chapter 3 require editing patterns with the goal of adding or removing particular program elements (types, fields, methods, or constructors) from the set matched by a pattern. Supposing we start with a pattern X,

- if we want to add an element, we synthesize a pattern p for that element, and replace X with $X \mid\mid p$; or
- if we want to remove an element, we synthesize a pattern p for that element, and replace X with X && $!p$.

This technique is simple—all we need is the ability to synthesize a pattern that selects exactly the element to add or remove, and the union, intersection, and negation operators on patterns. A pattern that selects exactly one element is easy to generate by using a fully-qualified name and signature. Unfortunately, the set operators can only be directly used on type patterns. That is, we can write

```
staticinitialization( Foo || Bar )
```

to add the type `Bar` to the type pattern `Foo`. But if we want to add a method `bar()` to the method pattern `void SomeClass.foo()`, we cannot write

```
call( void SomeClass.foo() || void SomeClass.bar() )
```

(this is a syntax error because the `call` pointcut takes a MethodPattern). To work around this problem in the case of pointcuts, we can make the following change instead:

```
call( void SomeClass.foo() )
```

$$\Downarrow$$

`call( void SomeClass.foo() )` || `call( void SomeClass.bar() )`

This is an example of a general technique. That is, if we have a pattern X appearing in a pointcut *pc*, and we want to achieve the effect of adding or removing a certain program element e from the set matched by the pattern, we can perform one of the following replacements. To add the element matched by pattern a:

$$\text{pc(X) =: pc(X) || pc(a)}$$

To remove the element matched by pattern a:

$$\text{pc(X) =: pc(X) \&\& !pc(a)}$$

In these replacements, the set operations in use apply to the pointcuts themselves, but the effect is equivalent to modifying only the pattern. Although it is not always literally possible, the refactorings in this thesis refer to broadening or narrowing patterns except when pointcuts are specifically of interest. Because the only kind of pattern that can occur outside of a primitive pointcut is a type pattern, between the set operations defined on type patterns and on pointcuts, this kind of narrowing or broadening is always possible.

4.2 Constituent Patterns

Identifier, type, arguments, and modifiers patterns are commonly used as constituent parts of field, method, and constructor patterns. Type patterns may also be used on their own to parameterize `this`, `target`, `args`, `staticinitialization`, `handler`, and `within` pointcuts.

Grammar

A grammar for AspectJ is presented in Appendix B, p. 77. The following extended BNF conventions are used:

Text in *'quotes'* represents literal text.

| in the right hand side of a production rule indicates a choice of possibilities.

+ denotes one or more occurrences.

$*$ denotes zero or more occurrences.

? denotes the preceding item as optional.

4.2.1 Identifier Patterns

$$\mathsf{IdentifierPattern} \longrightarrow (\mathsf{LETTER}|\mathsf{DIGIT}|'*')^{+}$$

Identifier patterns are the simplest of the patterns in AspectJ. They specify a simple regular expression that can match an identifier. The * symbol matches any identifier substring, so the identifier `refactor` would be matched by patterns including `*`, `refactor`, `ref*`, `*actor`, or `r*f*r*`.

Manipulating Identifier Patterns

As the simplest sub-patterns used in AspectJ, identifier patterns are atomic from the view of this thesis. Rather than manipulating them directly, we will manipulate higher-level patterns and, if a new identifier pattern needs to match a certain identifier, we will synthesize the most straightforward identifier pattern possible by using the identifier name itself.

4.2.2 Type Patterns

$$\begin{array}{lcl} \mathsf{TypePattern} & \longrightarrow & \mathsf{IdentifierPattern}\,((\text{'.'} \mid \text{'..'})\ \mathsf{IdentifierPattern})^{*}\ \text{'+'}?\ \text{'[ ]'}^{*} \\ & | & \text{'!'}\ \mathsf{TypePattern} \\ & | & \mathsf{TypePattern}\ \text{'\&\&'}\ \mathsf{TypePattern} \\ & | & \mathsf{TypePattern}\ \text{'||'}\ \mathsf{TypePattern} \\ & | & (\mathsf{TypePattern}) \end{array}$$

Type patterns match local or fully-qualified type names. The series of identifier patterns separated by dots matches a type name. The double-dot ('..') matches any string that begins and ends with a dot. A number of bracket-pairs ([ ]) can denote array types. The + symbol causes a type pattern to match all subtypes of all matching types. The ! symbol preceding a type pattern inverts the types that it matches. For combining type patterns, && is the intersection operator and || is the union operator.

Manipulating Type Patterns

To include or exclude a specific type from a type pattern, we can explicitly include or exclude the type by its fully qualified name. For example, if we want to exclude `org.rura.thesis.Foo` from a type pattern T, we can replace T with

$$T\ \&\&\ !\mathtt{org.rura.thesis.Foo}$$

Similarly, we can include `org.rura.thesis.Foo` in a type pattern T by replacing it with

$$T\ ||\ \mathtt{org.rura.thesis.Foo}$$

4.2.3 Arguments Patterns

$$\mathsf{ArgumentsPattern} \longrightarrow (\mathsf{ArgumentsPatternPart}\,(',' \,\mathsf{ArgumentsPatternPart})^{*})?$$

$$\mathsf{ArgumentsPatternPart} \longrightarrow (\mathsf{TypePattern}|'..')$$

Arguments patterns match the arguments of a method or constructor. Each part is either a type pattern, which matches the type of a parameter, or '..', which matches zero or more parameters. A method or constructor's parameters match an argument pattern if the argument pattern's parts can be matched in order to the method/constructor's parameters.

Manipulating Arguments Patterns

There isn't always a clear way to broaden or narrow an arguments pattern, since it depends on a combination of number of and types of parameters. Thus these are again treated as atomic.

An arguments pattern that selects a specific method's arguments can be synthesized by simply listing the types of that method's parameters in order.

4.2.4 Modifiers Patterns

$$\mathsf{ModifiersPattern} \longrightarrow ('!'?\ ('public'|'private'|'protected'|'static'|'final'|'strictfp'))^{*}$$

A modifiers pattern is used in a field, method, or constructor pattern to narrow the possible field/method/constructor matches to those whose declarations have (or do not have) certain modifiers. An empty modifiers pattern matches any declaration. Listing a specific modifier requires that the declaration have that modifier. Listing a modifier preceded by a ! requires that the declaration not have that modifier.

Manipulating Modifiers Patterns

A requirement for or against a specific modifier can be removed from a modifiers pattern by simple deleting it from the pattern, or added by appending it. However, none of the refactorings demand exactly this kind of change, so alterations are made at the higher-level patterns that contain modifiers patterns.

A declaration-specific modifier pattern can be made by listing the modifiers on a specific declaration.

4.3 Field, Method, and Constructor Patterns

These patterns pick out major program elements and are used to parameterize most of AspectJ's primitive pointcuts, such as `call`, `execution`, `get`, and `set` pointcuts.

For simplicity, the throws patterns used to narrow method and constructor patterns based on exceptions they can throw are not discussed here. The throws pattern is itself just a type pattern and can be handled in the same fashion as other type patterns within field, method, and constructor patterns.

4.3.1 Field Patterns

FieldPattern $\longrightarrow$ ModifiersPattern TypePattern (TypePattern '.')? IdentifierPattern

A field pattern matches a field declaration based on its signature. Parts of the signature are each matched to a sub-pattern:

- modifiers,
- the field's type,
- (optionally) the type containing the field, and
- the field's name.

Manipulating Field Patterns

A field pattern specific to a particular field can be synthesized by naming it explicitly using the following parts:

1. a TypePattern using the fully-qualified name of the field's type,
2. a TypePattern using the fully-qualified name of the field's containing type, and
3. an IdentifierPattern using the field's identifier.

A field pattern can be effectively expanded or shrunk using this synthesis approach with the pointcut-level technique described in section 4.1.

4.3.2 Method Patterns

MethodPattern $\longrightarrow$ ModifiersPattern TypePattern
(TypePattern '.')? IdentifierPattern '(' ArgumentsPattern ')'

A method pattern matches a method declaration based on its signature. Parts of the signature are each matched to a sub-pattern:

- modifiers,
- the method's return type,
- (optionally) the type containing the method,
- the method's name, and
- the method's arguments.

Manipulating Method Patterns

A method pattern specific to a particular method can be synthesized by naming the method explicitly using the following parts:

1. a TypePattern using the fully-qualified name of the method's return type,
2. a TypePattern using the fully-qualified name of the method's containing type,
3. an IdentifierPattern using the method's identifier, and
4. an ArgumentsPattern listing the fully-qualified type names of the method's parameters.

A method pattern can be effectively expanded or shrunk using this synthesis approach with the pointcut-level technique described in section 4.1.

4.3.3 Constructor Patterns

$$\mathsf{ConstructorPattern} \longrightarrow \mathsf{ModifiersPattern}\ (\mathsf{TypePattern}\ '.')?\ '\mathit{new}'\ '('\ \mathsf{ArgumentsPattern}\ ')'$$

A constructor pattern matches a constructor declaration based on its signature. Parts of the signature are each matched to a sub-pattern:

- modifiers,
- (optionally) the type containing the constructor, and
- the constructor's arguments.

A constructor pattern is essentially a method pattern without a return type and name.

Manipulating Constructor Patterns

A constructor pattern specific to a particular constructor can be synthesized by naming the method explicitly using the following parts:

1. a TypePattern using the fully-qualified name of the constructor's containing type, and
2. an ArgumentsPattern listing the fully-qualified type names of the constructor's parameters.

A constructor pattern can be effectively expanded or shrunk using this synthesis approach with the pointcut-level technique described in section 4.1.

Chapter 5

New Fundamental Refactorings for AOP

The low-level refactorings described in chapter 3 describe a basis for safely carrying out existing object-oriented refactorings on AspectJ programs. But new refactorings are also needed to help us take advantage of AOP features. This chapter describes new low-level aspect-oriented refactorings—refactorings that pertain specifically to AOP program elements. The next chapter builds on these low-level refactorings to describe systematic ways to abstract crosscutting concerns from existing code.

5.1 Aspects are Like Classes

Aspects can work like classes in many ways: they can contain methods and fields and extend superclasses (or superaspects). In fact, the only thing that a class can have but an aspect cannot is a (callable) constructor. Thus the most obvious set of aspect-specific refactorings are simply straightforward analogs of refactorings that apply to classes or members. These include:

1. Create Field, §3.1.3
2. Create Method, §3.1.4
3. Delete Unreferenced Field, §3.2.2
4. Delete Set of Unreferenced Methods, §3.2.3
5. Rename Class (becomes Rename Aspect), §3.3.1
6. Rename a Variable, §3.3.2
7. Rename a Method, §3.3.3

8. Change Type, §3.3.4
9. Change Access Control Mode, §3.3.5
10. Add Parameter to Method, §3.3.6
11. Remove Unreferenced Method Parameter, §3.3.7
12. Reorder Method Parameters, §3.3.8
13. Add Method Body (Concretize Abstract Method), §3.3.9
14. Remove Method Body (Make Concrete Method Abstract), §3.3.10
15. Replace Field References With Accessor/Modifier Calls, §3.3.11
16. Replace Statement List with Method Call, §3.3.12
17. Inline Method Call, §3.3.13
18. Change a Class's Superclass (becomes Change an Aspect's Superaspect or Superclass), §3.3.14
19. Move Field to Superclass (becomes Move Field to Superaspect), §3.4.1
20. Move Field to Subclasses (becomes Move Field to Subaspects), §3.4.2

5.2 Creating a Program Entity

5.2.1 Create Empty Aspect

Define a new aspect with no locally defined members, optionally with a designated superaspect.

Preconditions:

1. The aspect's name does not conflict with an already existing type.
2. ✂ The aspect must not be subject to a combination of inter-type parent or member declarations that will cause it to fail compilation.

These preconditions guarantee that the new aspect will compile: it will have a unique name (L2), will follow rules about `extends` and `implements` declarations (L7), and will not include illegal members (L3, L4, L6, L8).

Because a new aspect is never referenced and contains no advice or inter-type declarations, the behavior of the program does not change when it is added.

5.2.2 Create Introduced Field or Method

Given a type pattern, add an unreferenced field or method to the set of types matching that pattern using static introduction from an aspect. Each introduction into a type has the same preconditions and effects on program behavior as a local addition (§3.1.3, 3.1.4).

Preconditions for each target type:

- If creating a field:
    1. The field's name does not conflict with an existing locally defined, inherited, or introduced field.
    2. If the field is not declared with private visibility, no name conflict occurs in any subclasses of the target class.
- If creating a method:
    1. The new method will compile as a member of the target class.
    2. If the new method will overload an existing method (either in the target class or in its subclasses), it must either be more general (thus not capturing any calls that would have invoked the existing method) or more precise and semantically equivalent to the method it overloads.
    3. If the target class has an inherited method that will be overridden by the new method, either that method is unreferenced on the target class and its subclasses, or the new method is semantically equivalent (e.g. identical) to the method it overrides.

We must verify these preconditions for every target type into which a member is introduced. For an argument that these preconditions guarantee behavior preservation in a single class, see sections 3.1.3 and 3.1.4 on p. 34.

5.2.3 Create Named Pointcut

Add a named pointcut declaration to a type.

Preconditions:

1. The pointcut's name must not conflict with another member in the containing type.
2. The pointcut expression assigned to the given name must compile in the containing type.
3. If the pointcut's name is the same as an inherited pointcut, either that inherited pointcut is unreferenced on the target type and its subtypes, or the new pointcut is semantically equivalent to the pointcut it overrides.

4. If a subtype of the target type defines a pointcut with the same name, this new pointcut declaration has the same or narrower visibility than the declaration in a subtype.

Creating a pointcut declaration merely gives a name to a set of joinpoints without associating any new behavior with them. Program behavior is not affected.

5.2.4 Create Empty Advice

Add a `before` or `after` advice declaration to an aspect with an empty body, or add an `around` advice declaration with a body containing only a call to `proceed()`. The advice may be parameterized by any pointcut expression.

Preconditions:

1. The advice's pointcut expression must compile in the aspect containing the advice.

In the case of before or after advice, regardless of what pointcut is specified, an empty advice body will cause nothing different to happen before or after the included joinpoints. Similarly, around advice will immediately proceed to the included joinpoints.

5.3 Deleting a Program Entity

5.3.1 Delete Unreferenced and Empty Aspect

Remove an aspect that is unreferenced and contains no advice bodies or inter-type declarations.

Preconditions:

1. There are no references to locally-defined or inherited members on instances of the aspect.
2. The aspect contains no inter-type member introductions.
3. The aspect contains no inter-type `parents`, `error`, or `warning` declarations.
4. The aspect contains no advice bodies.

Like a class, if an aspect is unreferenced, none of its members will be called. Unlike a class, however, an aspect can affect program behavior without being directly referenced if it contains inter-type declarations or advice bodies.

5.3.2 Delete Unreferenced Introduced Field

Remove an inter-type field declaration that is not referenced on any of its target types.

Preconditions:

1. The field is not referenced on any of its target types (or their subtypes). Some ways this condition could be reached are:

- the type pattern that matches target types actually matches nothing, so the field is in fact not introduced anywhere,
- all of the target types matching the type pattern are interfaces or abstract classes that are never concretized by an implementing class, or
- **most generally,** the field is known to be unreferenced on each type to which it is introduced and its subtypes.

Since the field is never referenced, no joinpoints associated with it will ever be reached in the program. Thus removing it does not change program behavior.

5.3.3 Delete Set of Unreferenced Introduced Methods

Remove a set of inter-type method declarations that share a common target type pattern.

Preconditions:

1. Each method to be deleted is

   - not referenced from outside the set of methods to be deleted (in any of the types matching the type pattern),
   - *or* redundant because a semantically equivalent method is inherited (in every matching type),
   - *or* redundant because it overloads a semantically equivalent method with a more general signature (in every matching type).

As with an unreferenced field, no joinpoints associated with these methods will ever be reached in the program. If a semantically equivalent method is inherited, removing a locally defined method simply causes that inherited method to be invoked instead. If a semantically equivalent method is overloaded and we remove the more specific method, the more general method will simply be invoked in its place. Thus if one of these conditions is satisfied for each member into which a method is introduced, removing the introduction does not change program behavior.

5.3.4 Delete Unreferenced Named Pointcut

Remove a named pointcut declaration that is not referenced.

Preconditions:

1. The pointcut is never referenced (its initialization with a pointcut expression does not count as a reference).

If the named pointcut is never referenced, it cannot affect program behavior and removing its declaration will not prevent the program from compiling.

5.3.5 Delete Unreachable Advice

Remove an advice declaration that can never be invoked because its pointcut contains no joinpoints that can occur. In general, it is impossible to tell whether a given pointcut will be empty, but in numerous cases (such as when the method pattern of a `call` pointcut matches no methods) it is possible to know that a pointcut is empty.

Preconditions:

1. The pointcut parameterizing the advice declaration is empty.

5.3.6 Delete Empty Advice

Remove an advice declaration that does not execute any code. The advice may be parameterized by any pointcut expression.

Preconditions:

1. If removing a `before` or `after` advice declaration, the body is empty.
2. If removing an `around` advice declaration, the body contains only a call to `proceed()`.

For any kind of advice that meets these preconditions, no code will be executed if it is invoked, and thus it does not currently affect program behavior.

5.4 Changing a program entity

5.4.1 Rename an Introduced Member

Change the name of an introduced field or method. This has the same preconditions and effects on program behavior as a local renaming (§3.3.2 and 3.3.3, p. 37), but generalized to all targets of introduction.

5.4.2 Rename a Named Pointcut

Change the name of a declared pointcut and all references (including in subtypes).

Preconditions:

1. The new name doesn't conflict with an already existing member in the containing type.
2. If the new name is the same as an inherited pointcut, either that inherited pointcut is unreferenced on the target type and its subtypes, or the new pointcut is semantically equivalent to the pointcut it overrides.
3. Either the pointcut is `private`, or no subtype of the target type defines a pointcut with the same name as the new name.

5.4.3 Add Parameter to Introduced Method

Identical to adding a parameter to a locally declared method (§3.3.6), but the preconditions must be satisfied for all types matching the introduction target type pattern.

5.4.4 Remove Unreferenced Parameter from Introduced Method

Identical to removing a parameter from a locally declared method (§3.3.7), but the preconditions must be satisfied for all types matching the introduction target type pattern.

5.4.5 Reorder Parameters of Introduced Method

Identical to reordering parameters of a locally declared method (§3.3.8), but the preconditions must be satisfied for all types matching the introduction target type pattern.

5.4.6 Merge Introductions

Replace two inter-type member introductions in an aspect, differing only in their target patterns, with a single introduction. If the original introductions have target patterns A and B, the introduction that replaces them will have target pattern $A||B$.

Since the introduced member is itself equivalent, changing the way it is introduced cannot affect program behavior, since both introductions are in the same aspect.

5.5 Moving Program Elements

5.5.1 Move Local Member Declaration to Introduction from Aspect

Move the definition of a member from within a type to an aspect. The declaration is changed to specify the original type as the introduction target explicitly by using its unambiguous name.

Preconditions:

1. The member must be public.

The precondition keeps the introduction simple because protected introduction is not allowed and private introduction means private to the containing aspect.

5.5.2 Move Member from Aspect to Local Declaration

Move the definition of a member from introduction from an aspect into a local declaration in each type that matches the introduction pattern.

There are no preconditions; if an inter-type member introduction is legal, the local declarations must be legal in each target type.

5.5.3 Move Supertype Declaration from Local to Aspect

Remove an `extends` clause with a supertype *s* from a type *c*, replacing it with a `declare parents:` *c* `extends` *s* statement. An analogous inter-type declaration can also be used to replace an `implements` clause.

Since the declaration is legal locally, it will also be legal as an inter-type declaration.

5.5.4 Localize Inter-type `parents` Declaration

Given an inter-type `parents` declaration of the form `declare parents:` *c* `extends` *s*, remove that declaration and change the locally declared superclass of *c* to be *s*. An analogous local declaration can also be used to replace an inter-type `implements` clause.

Since the inter-type supertype declaration is legal, it will also be legal locally.

5.5.5 Move Advice

Move an advice declaration from one aspect to another.

1. Any references to named pointcuts, variables, or patterns used to parameterize the advice must resolve identically in the new aspect.
2. The advice body must compile in the new aspect.
3. The advice body must be semantically equivalent in its new context; that is, all names must resolve identically after the move.

5.6 Altering Advice

5.6.1 Merge Advice Bodies

Given two before or after advice bodies with the same parameters, combine them by moving one's body block into the other's body block.

Preconditions:

1. The order of execution of these two advice bodies with respect to the same joinpoints does not affect program behavior, or is well defined statically.

5.6.2 Generalize `before` or `after` Advice to around Advice

Change `before` or `after` advice to equivalent `around` advice. To do this, change the advice type to `around` and

- in the case of `before` advice, add a `proceed()` statement at the end of the advice body;
- in the case of `after` advice, add a `proceed()` statement at the beginning of the advice body.

Because the same code occurs at the same time relative to each joinpoint, behavior is not affected.

5.6.3 Inline Advice on Method Call

Inline the body of advice `before`, `after`, or `around`, a `call` pointcut, by changing method call sites as follows for each class c in which a matching call occurs.

For each matching method m called from c, create in c a helper method m' with the same arguments and return type as m. m' will contain the inlined version of the call advice.

- For `before` advice:

```
ReturnType  m'(a_1, ..., a_n) {
                advice body
                return m(a_1, ..., a_n);
}
```

- For `after` advice:

```
ReturnType  m'(a_1, ..., a_n) {
                ReturnType result = m(a_1, ..., a_n);
                advice body
                return result;
}
```

- For `around` advice:

```
ReturnType  m'(a_1, ..., a_n) {
                advice body before proceed
                advice body with proceed() replaced by m(a_1, ..., a_n)
                advice body after proceed, including a return statement
}
```

Preconditions:

1. The actual names of arguments $a_1, \ldots, a_n$, the `result` variable, and the methods m' are selected such that they are legal and not shadowed by declarations in the advice body.
2. All references from within the advice body resolve equivalently from the scope of m'.
3. The call pointcut is not intersected with `within` or `withincode` pointcuts.

Provided variable names are not captured (precondition 1) and advice bodies actually can be inlined (precondition 2) the transformation technique described guarantees that the advice code is executed equivalently. The final precondition guarantees that advice execution is not contingent on where the method is called, thus guaranteeing that the call can be safely moved into our new helper method.

Chapter 6

High-Level AOP Refactorings

High-level refactorings are composite refactorings built from fundamental refactorings such as those described in chapters 3 and 5. Although low-level refactorings are often useful on their own, these refactorings address more complex and specific design problems. In particular, the high-level refactorings described in this chapter are intended to aid in the extraction of crosscutting concerns by deploying AOP techniques in existing programs.

6.1 Move Static Introduction

To move a static inter-type member introduction to a different aspect, we can inline (localize) it from the original aspect into its target types (§5.5.2) and then extract it into the desired target aspect (§5.5.1).

This is an example of a refactoring that accomplishes a simple but potentially useful change and can be composed of the fundamental refactorings described in the previous chapter. In a refactoring tool, this would likely be implemented as their own primitive refactorings; the purpose of listing it here is to show that it can be achieved.

6.2 Extract Common Members to Aspect

Given a set of classes, extract common members into a new aspect.

- Create a new aspect a (§5.2.1).
- Select a set C of classes.
- Create in the aspect a new, empty interface (§3.1.2). This will be used to group the selected classes. Add `declare parents: implements` statements so that each class implements this empty interface (§3.3.15).

- Find a group of members that is semantically equivalent across the classes. Perform the *Move Local Member Declaration to Introduction from Aspect* refactoring (§5.5.1) for each member, moving the definition into the new aspect.

6.3 Extract Interface Support into Aspect

Given a class and an interface that it implements, move all features responsible for supporting that interface into a new aspect. (Move only those members required by the selected interface and not by any of the other interfaces the class may implement.)

- Select a class c and an interface i that it implements.
- Let M be the set of members of c that implement members in i and do not override or implement any other members from other supertypes of c.
- Create a new aspect a (§5.2.1).
- Perform the *Move Supertype Declaration from Local to Aspect* refactoring, §5.5.3, on the `implements` clause for i.
- For each member in M, perform the *Move Local Member Declaration to Introduction from Aspect*, §5.5.1 to move its declaration into the new aspect.

6.4 Extract Disjoint State into Aspect

Select a group of members in a class that are disjoint from other members in the class. That is, select a set of members $D \subseteq M$ where:

- M is the set of all members in the class
- $\text{referencedFrom}(X, m)$ is true iff the member m is referenced from within a member of set X
- $\forall d \in D$

$$\begin{aligned} &\neg\ \text{referencedFrom}(M \setminus D, d) \\ &\wedge\ \forall m \in M \setminus D, \neg\text{referencedFrom}(D, m) \end{aligned}$$

Then, for each member in D, perform the *Move Local Member Declaration to Introduction from Aspect*, §5.5.1, to move its declaration into the new aspect.

Chapter 7

Conclusions

This thesis lays some of the foundations for incorporating refactoring into aspect-oriented software development (AOSD). Because refactoring and AOP can contribute in different ways to better-designed software, combining them can compound their value. The refactorings presented here address many potential design changes. Still, work in this area is only beginning, experience in AOSD practice is small, and tools are lacking. Much additional work is required before these or other refactorings can become accessible tools for the AOP developer.

7.1 Summary of Contributions

This thesis takes Opdyke's [Opd92] set of low-level refactorings and reconsiders each in the context of aspect-oriented programming. It also defines a new set of low-level and composite refactorings that operate on aspect-oriented software. These entail the following steps:

1. A simplifying assumption about overloading and inheritance is removed. (Opdyke avoided overloading and simplified inheritance by allowing no more than one method with the same name in any class.) This thesis addresses the full range of inheritance and overloading possibilities in AspectJ.

2. In cases where C++ (the original basis for the refactorings) differs from Java (and thus AspectJ), refactorings and constraints are converted from C++ to Java variants. This is most clearly manifested in the language requirements: for example, Java does not allow overriding of inherited fields while C++ does. This thesis also generalizes refactorings to include interfaces, a feature that does not exist in C++.

3. The preconditions, steps, and behavior preservation arguments of existing refactorings are extended to guarantee behavior preservation in AspectJ.

4. A set of new refactorings that apply specifically to aspect-oriented programming are defined. For each of these, preconditions and required steps are given, and the transformation is argued to be behavior preserving.

7.2 Limitations of Approach

There are several limitations and assumptions in the approach of this thesis:

1. A large subset of AspectJ, but not the whole language, is discussed. Aspect cardinality, `error` and `warning` declarations, and throws patterns (see Special aspect declarations, p. 19) are not addressed, because they are infrequently used language features that would add complexity to the explanations in this thesis. (No simplifying assumptions are made regarding Java.)

2. Refactorings attempt to avoid changing the meaning of dynamic pointcuts, but the emphasis of these refactorings is on static program transformation. While there may be better ways, such as dynamic program analysis, to guarantee behavior preservation when dynamic pointcuts are in use, this thesis does not address these possibilities.

3. The behavior of programs using reflection is not generally preserved.

7.3 Future Work

Aspect-oriented refactoring derives from research across multiple programming language and software engineering disciplines, including program analysis and transformation, aspect-oriented programming and separation of concerns, and actual software development experience. Specific areas for future work include automatic refactoring tools, new refactorings, better analyses to allow refactorings in more cases, and real-world evaluation.

7.3.1 Automated Refactoring Tools

For refactoring to become a truly viable and efficient tool for software engineers, automated support is needed. Many of the analyses and transformations required are not complicated, but are time-consuming and error-prone when executed by hand. An automatic refactoring tool can help a programmer to experiment with a variety of program designs, and nearly eliminates the cost of refactoring, making it convenient to refactor whenever any design improvement is possible.

Aspect Mining

One task that must precede refactoring is deciding what parts of a program to refactor. Especially with crosscutting concerns, these parts may be spread throughout a program and difficult to uncover, especially in complex and unfamiliar programs. Tools can assist in this task by exploiting lexical, stylistic, and semantic features of source code. Existing work in this area includes the Aspect Mining Tool [HK01] and the Aspect Browser [Gri98]. Though these tools are important first steps, the simple textual or semantic analyses they perform will likely be superseded by more sophisticated, higher-level analyses as AOP and refactoring experience matures.

7.3.2 Refactoring in the Real World

Studying the long-term development of commercial aspect-oriented software systems can benefit AOP refactoring research in two ways: we can begin to understand how known refactorings are really used, and we can observe and formalize *ad hoc* refactorings. A better understanding of how AOP refactorings are used would illuminate the limitations of these refactorings, such as cases where the conservative assumptions we make in order to guarantee meaning preservation are too constraining. This could lead to improved analysis techniques or a more flexible theoretical foundation for behavior preservation arguments.

Major prior works on refactoring have taken the approach of describing refactorings observed in actual software development experiences either as a process of reflection by the author [Fow00] or by a more disciplined study of evolution in specific systems [Opd92]. In both cases, experience illustrated the refactorings; the authors then described the transformations they found most useful. This thesis concentrates largely on small transformations that seem to be fundamental in most larger refactorings. But specifically for higher-level refactorings, experience can lead to important new discoveries.

For either of these undertakings, a larger body of experience in aspect-oriented software development is required.

7.3.3 A Formal Framework for AOP and Refactoring

Because AOP and Refactoring are both newcomers to scientific research, there is not yet a widely accepted formal framework for discussing either the structure of AOP software, nor the analyses and transformations of refactoring. As tools are developed and refined, standard formal frameworks can help by enabling programmers to define and share refactorings. Another useful application of a formal framework would be in support of proofs that refactorings are behavior preserving.

7.3.4 The Very Big Picture: Integration of Software Engineering Tools and Techniques

This thesis brings together two disciplines whose common goal is to equip the software engineer to achieve better design. These, however, are just two approaches out of many that span the software life cycle. In particular, in recent years software engineers have seen a proliferation of different *models* of their software systems—UML structure and interaction diagrams, architectural models, version control systems that track software changes, and team-oriented systems that divide programs based on a who is responsible for different parts of the code. Each of these is an important departure from the traditional view of a program as having one just one editable form (source code) and one runnable form. Refactoring is one way of tightening a loop: rather than a unidirectional flow from design to implementation, the design of already working code can be improved in-place. But opportunities for this kind of unification exist between many software engineering approaches and tools. I believe that this kind of integration, on a large scale, is our best chance to give software engineers the kind of flexibility in working with their systems that is commonly enjoyed by architects and other designers. This is a considerable effort, but may make possible an integrated software engineering tool set that is greater than the sum of its parts.

Bibliography

[B^+01] Kent Beck et al. The Agile Software Manifesto. Website, 2001. Available at `http://agilemanifesto.org/`.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, Upper Saddle River, NJ, 1997.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Com02] AOSD Steering Committee. Aspect-Oriented Software Development: Tools & Languages. Website, 2002. Available at `http://aosd.net/tools.html`.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[Fow02] Martin Fowler. Refactoring Tools. Website, 2002. Available at `http://www.refactoring.com/`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition edition, 2000.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[Goo02] Google Web Directory. Refactoring Tools. Website, 2002. Available at `http://directory.google.com/Top/Computers/Programming/Methodologies/Refactoring/Tools/`.

[Gri91] W. G. Griswold. *Program restructuring as an aid to software maintenance*. PhD thesis, University of Washington, 1991.

[Gri98] W. G. Griswold. Coping with software change using information transparency. Technical Report CS98-585, Department of Computer Science and Engineering, University of California, San Diego, August 1998.

[HK01] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition of legacy code. In *International Conference on Software Engineering*, Toronto, 2001. Workshop on Advanced Separation of Concerns.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java & AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, Budapest, Hungary, 2001. Springer. Available from `http://www.aspectj.org`.

[Mic68] Donald Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, 1968.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible framework for AOP in Java. In *Reflection*, Kyoto, Japan, 2001. JAC website: `http://jac.aopsys.com/`.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[Tea02a] The AspectJ Team. *AspectJ API Documentation.* Xerox Corporation, 2002. Available from `http://www.aspectj.org`.

[Tea02b] The AspectJ Team. *The AspectJ Programming Guide.* Xerox Corporation, 2002. Available from `http://www.aspectj.org`.

[Tea03] The AspectJ Team. *AspectJ Quick Reference.* Xerox Corporation, 2003. Available from `http://www.aspectj.org`.

[TOHJ99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, 1999. Hyper/J website: `http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`.

Appendix A

AspectJ Quick Reference

The AspectJ 1.1b2 language reference [Tea03] is reproduced on the next two pages.

AspectJ Quick Reference

Aspects — *at top-level or* **static** *in types*

aspect *A* { … }
defines the aspect *A*

privileged aspect *A* { … }
A can access private fields

aspect *A* **extends** *B* { … }
B is a class or abstract aspect
aspect *A* **implements** *B* { … }
B is an interface

general form:
[**privileged**] [*Modifiers*]
aspect *Id*
[**extends** *Type*]
[**implements** *TypeList*]
{ *Body* }

Pointcut definitions — *in types*

private pointcut *pc*() : *call(void Foo.m())* ;
a pointcut visible only from the defining type
pointcut *pc*(*int i*) : *set(int Foo.x) && args(i)* ;
a package-visible pointcut that exposes an *int.*

public abstract pointcut *pc*() ;
an abstract pointcut that can be referred to from anywhere.
abstract pointcut *pc*(*Object o*) ;
an abstract pointcut visible from the defining package. Any pointcut that implements this must expose an *Object.*

general form:
abstract [*Modifiers*] **pointcut** *Id* (*Formals*) ;
[*Modifiers*] **pointcut** *Id* (*Formals*) : *Pointcut* ;

Advice declarations — *in aspects*

before () : *get(int Foo.y)* { ... }
runs before reading the field *int Foo.y*

after () **returning** : *call(int Foo.m(int))* { ... }
runs after calls to *int Foo.m(int)* that return normally.
after () **returning** (*int x*) : *call(int Foo.m(int))* { ... }
same, but the return value is named *x* in the body.

after () **throwing** : *call(int Foo.m(int))* { ... }
runs after calls to *int Foo.m(int)* that return abruptly by throwing an exception.
after () **throwing** (*NotFoundException e*) :
call(int Foo.m(int)) { ... }
runs after calls to *int Foo.m(int)* that return abruptly by throwing a NotFoundException. The thrown exception is named *e* in the body.

after () : *call(int Foo.m(int))* { ... }
runs after calls to *int Foo.m(int)* regardless of how returned

before(*int i*) : *set(int Foo.x) && args(i)* { ... }
runs before field assignment to *int Foo.x*. The value to be assigned is named *i* in the body
before(*Object o*) : *set(* Foo.*) && args(o)* { ... }
runs before field assignment to any field of *Foo*. The value to be assigned is converted to an object type (*int* to *Integer,* for example) and named *o* in the body
int **around** () : *call(int Foo.m(int))* { ... }
runs instead of calls to *int Foo.m(int)*, and returns an *int*. In the body, continue the call by using **proceed**(), which has the same signature as the around advice.
int **around** () **throws** *IOException* :
call(int Foo.m(int)) { ... }
same, but the body is allowed to throw *IOException*
Object **around** () : *call(int Foo.m(int))* { ... }
same, but the value of **proceed**() is converted to an *Integer*, and the body should also return an *Integer* which will be converted into an *int*

general form:
[**strictfp**] *AdviceType* : *Pointcut* { *Body* }
where *AdviceType* is one of
before (*Formals*)
after (*Formals*)
after (*Formals*) **returning** [(*Formal*)]
after (*Formals*) **throwing** [(*Formal*)]
Type **around** (*Formals*) [**throws** *TypeList*]

Special forms — *in advice*

thisJoinPoint
reflective information about the join point.
thisJoinPointStaticPart
the equivalent of **thisJoinPoint.getStaticPart()**, but may use fewer resources.
thisEnclosingJoinPointStaticPart
the static part of the join point enclosing this one.

proceed (*Arguments*)
only available in **around** advice. The *Arguments* must be the same number and type of the parameters of the advice.

Inter-type Member Declarations — *in aspects*

int Foo . m (*int i*) { ... }
a method *int m(int)* owned by *Foo*, visible anywhere in the defining package. In the body, **this** refers to the instance of *Foo*, not the aspect.
private *int Foo . m* (*int i*) **throws** *IOException* { ... }
a method *int m(int)* that is declared to throw *IOException,* only visible in the defining aspect. In the body, **this** refers to the instance of *Foo*, not the aspect.

abstract *int Foo . m* (*int i*) ;
an abstract method *int m(int)* owned by *Foo*

Point . **new** (*int x, int y*) { ... }
a constructor owned by *Point.* In the body, **this** refers to the new *Point*, not the aspect.
private static *int Point . x* ;
a static *int* field named *x* owned by *Point* and visible only in the declaring aspect
private *int Point . x = foo()* ;
a non-static field initialized to the result of calling *foo()*. In the initializer, **this** refers to the instance of *Foo*, not the aspect.

general form:
[*Modifiers*] *Type Type . Id* (*Formals*)
[**throws** *TypeList*] { *Body* }
abstract [*Modifiers*] *Type Type . Id* (*Formals*)
[**throws** *TypeList*] ;
[*Modifiers*] *Type* . new (*Formals*)
[**throws** *TypeList*] { *Body* }
[*Modifiers*] *Type Type . Id* [= *Expression*] ;

Other Inter-type Declarations _in aspects_

declare parents : _C_ **extends** _D_;
declares that the superclass of _C_ is _D_. This is only legal if _D_ is declared to extend the original superclass of _C_.
declare parents : C **implements** _I, J_ ;
_C_ implements _I_ and _J_

declare warning : _set(* Point.*) && !within(Point)_ :
"bad set" ;
the compiler warns _"bad set"_ if it finds a set to any field of _Point_ outside of the code for _Point_
declare error : _call(Singleton.new(..))_ :
"bad construction" ;
the compiler signals an error _"bad construction"_ if it finds a call to any constructor of _Singleton_

declare soft : _IOException_ :
execution(Foo.new(..));
any IOException thrown from executions of the constructors of _Foo_ are wrapped in **org.aspectj.SoftException**

declare precedence : _Security, Logging, *_ ;
at each join point, advice from _Security_ has precedence over advice from _Logging_, which has precedence over other advice.

general form
declare parents : _TypePat_ **extends** _Type_ ;
declare parents : _TypePat_ **implements** _TypeList_ ;
declare warning : _Pointcut_ : _String_ ;
declare error : _Pointcut_ : _String_ ;
declare soft : _TypePat_ : _Pointcut_ ;
declare precedence : _TypePatList_ ;

Primitive Pointcuts

call (_void Foo.m(int)_)
a call to the method _void Foo.m(int)_
call (_Foo.new(..)_)
a call to any constructor of _Foo_
execution (_* Foo.*(..) throws IOException_)
the execution of any method of _Foo_ that is declared to throw _IOException_
execution (_!public Foo .new(..)_)
the execution of any non-public constructor of _Foo_

initialization (_Foo.new(int)_)
the initialization of any _Foo_ object that is started with the constructor _Foo(int)_
preinitialization (_Foo.new(int)_)
the pre-initialization (before the **super** constructor is called) that is started with the constructor _Foo(int)_
staticinitialization(_Foo_)
when the type _Foo_ is initialized, after loading
get (_int Point.x_)
when _int Point.x_ is read
set (_!private * Point.*_)
when any non-private field of _Point_ is assigned
handler (_IOException+_)
when an _IOException_ or its subtype is handled with a catch block
adviceexecution()
the execution of all advice bodies

within (_com.bigboxco.*_)
any join point where the associated code is defined in the package _com.bigboxco_
withincode (_void Figure.move()_)
any join point where the associated code is defined in the method _void Figure.move()_
withincode (_com.bigboxco.*.new(..)_)
any join point where the associated code is defined in any constructor in the package _com.bigoxco._

cflow (_call(void Figure.move())_)
any join point in the control flow of each call to _void Figure.move()_. This includes the call itself.
cflowbelow (_call(void Figure.move())_)
any join point below the control flow of each call to _void Figure.move()_. This does not include the call.

if (_Tracing.isEnabled()_)
any join point where _Tracing.isEnabled()_ is **true**. The boolean expression used can only access static members, variables bound in the same pointcut, and **thisJoinPoint** forms.

this (_Point_)
any join point where the currently executing object is an instance of _Point_
target (_java.io.InputPort_)
any join point where the target object is an instance of _java.io.InputPort_
args (_java.io.InputPort, int_)
any join point where there are two arguments, the first an instance of _java.io.InputPort,_ and the second an _int_

args (_*, int_)
any join point where there are two arguments, the second of which is an _int._
args (_short, .., short_)
any join point with at least two arguments, the first and last of which are _short_s

any position in **this**, **target**, and **args** can be replaced with a variable bound in the advice or pointcut.

general form:
call(_MethodPat_)
call(_ConstructorPat_)
execution(_MethodPat_)
execution(_ConstructorPat_)
initialization(_ConstructorPat_)
preinitialization(_ConstructorPat_)
staticinitialization(_TypePat_)
get(_FieldPat_)
set(_FieldPat_)
handler(_TypePat_)
adviceexecution()

within(_TypePat_)
withincode(_MethodPat_)
withincode(_ConstructorPat_)

cflow(_Pointcut_)
cflowbelow(_Pointcut_)

if(_Expression_)

this(_Type_ | _Var_)
target(_Type_ | _Var_)
args(_Type_ | _Var_ , …)
where
MethodPat:
[_ModifiersPat_] _TypePat_ [_TypePat_ .] _IdPat_ (_TypePat_ , …)
[**throws** _ThrowsPat_]
ConstructorPat:
[_ModifiersPat_] [_TypePat_ .] **new** (_TypePat_ , …)
[**throws** _ThrowsPat_]
FieldPat:
[_ModifiersPat_] _TypePat_ [_TypePat_ .] _IdPat_
TypePat:
IdPat [**+**] [**[]** …]
! _TypePat_
TypePat **&&** _TypePat_
TypePat **||** _TypePat_
(_TypePat_)

Appendix B

AspectJ Grammar

The following grammar describes the AspectJ 1.0 language (thereby including Java programs). This thesis does not discuss all features of the language.

The starting symbol of each file input to the AspectJ compiler is the CompilationUnit.

Notation	This grammar uses the following extended BNF conventions:
	Text in *'quotes'* represents literal text.
	| **in the right hand side of a production rule** indicates a choice of possibilities.
	+ denotes one or more occurrences.
	∗ denotes zero or more occurrences.
	? denotes the preceding item as optional.

Type Declaration

AspectJ allows aspects to be declared at the top level (in a CompilationUnit) or `static` within types.

```
TypeDeclaration  ⟶  ClassDeclaration
                  |  InterfaceDeclaration
                  |  AspectDeclaration
```

Aspect Declaration

AspectDeclaration	$\longrightarrow$	(Modifier)* UnmodifiedAspectDeclaration
NestedAspectDeclaration	$\longrightarrow$	$'static'$ (Modifier)* UnmodifiedAspectDeclaration
UnmodifiedAspectDeclaration	$\longrightarrow$	$'aspect'$ Name
		($'extends'$ Name)?
		($'implements'$ Name (',' Name)*)?
		($'dominates'$ Name (',' Name)*)?
		'{'
		(AspectBodyDeclaration)*
		'}'
AspectBodyDeclaration	$\longrightarrow$	Initializer
	|	NestedClassDeclaration
	|	NestedInterfaceDeclaration
	|	MethodDeclaration
	|	NestedAspectDeclaration
	|	PointcutDeclaration
	|	AdviceDeclaration
	|	InterTypeMethodDeclaration
	|	InterTypeFieldDeclaration
	|	InterTypeDeclare

Aspect Members

AspectBodyDeclaration	$\longrightarrow$	Initializer
	|	PointcutDeclaration
	|	AdviceDeclaration
	|	InterTypeMethodDeclaration
	|	InterTypeFieldDeclaration
	|	InterTypeDeclare

Pointcuts

Named pointcuts may be declared in any type (class, aspect, or interface).

PointcutDeclaration	$\longrightarrow$	(Modifier)* '*pointcut*' FormalParameters
		(' :' PointcutExpression)?
	|	PointcutDeclaration

PointcutExpression	$\longrightarrow$	PointcutExpression && PointcutExpression
	|	PointcutExpression || PointcutExpression
	|	'!' PointcutExpression
	|	'('PointcutExpression')'
	|	Identifier Arguments
	|	PrimitivePointcut

PrimitivePointcut	$\longrightarrow$	('*call*' | '*execution*' | '*initialization*' | '*withincode*')
		'(' ConstructorPattern ')'
	|	('*call*' | '*execution*' | '*withincode*') '(' MethodPattern ')'
	|	('*staticinitialization*' | '*handler*' | '*within*') '(' TypePattern ')'
	|	('*this*' | '*target*' | '*args*') '(' TypePattern | Identifier ')'
	|	('*cflow*' | '*cflowbelow*') '(' PointcutExpression ')'
	|	'*if*' '(' Expression ')'

Advice

AdviceDeclaration	$\longrightarrow$	(Modifier)* AdviceType ' :'
		PointcutExpression Block

AdviceType	$\longrightarrow$	'*before*' FormalParameters
	|	'*after*' FormalParameters '*returning*' ('('FormalParameter')')?
	|	'*after*' FormalParameters '*throwing*' ('('FormalParameter')')?
	|	'*after*' FormalParameters
	|	ReturnType '*around*' FormalParameters ('*throws*' NameList)?

Inter-Type Declarations

Field Introduction

InterTypeFieldDeclaration $\longrightarrow$ (Modifier)* Type InterTypeVariableDeclarator
(InterTypeVariableDeclarator)*

InterTypeVariableDeclarator $\longrightarrow$ TypePattern $'.'$ Identifier ($'['$ $']'$)*
($'='$ VariableInitializer)

Method Introduction

InterTypeMethodDeclaration $\longrightarrow$ (Modifier)*
ReturnTypeTypePattern $'.'$ Identifier FormalParameters ($'['$ $']'$)*
($'throws'$ NameList)? (Block)?

Other Inter-type Declarations

InterTypeDeclare $\longrightarrow$ $'declare'$ $'parents'$ $':'$
TypePattern ($'extends'$ Name | $'implements'$ NameList)
| $'declare'$ ($'warning'$ | $'error'$) $':'$
PointcutExpression $':'$ StringLiteral
| $'declare'$ $'soft'$ $':'$ TypePattern $':'$ PointcutExpression

AspectJ Patterns

```
       IdentifierPattern  ⟶  (LETTER|DIGIT|'*')+

             TypePattern  ⟶  IdentifierPattern (('.' | '..') IdentifierPattern)* '+'? '[ ]'*
                           |  '!' TypePattern
                           |  TypePattern '&&' TypePattern
                           |  TypePattern '||' TypePattern
                           |  (TypePattern)

        ArgumentsPattern  ⟶  (ArgumentsPatternPart (',' ArgumentsPatternPart)*)?

    ArgumentsPatternPart  ⟶  (TypePattern|'..')

        ModifiersPattern  ⟶  ('!'? Modifier)*

            FieldPattern  ⟶  ModifiersPattern TypePattern
                              (TypePattern '.')? IdentifierPattern

           MethodPattern  ⟶  ModifiersPattern TypePattern
                              (TypePattern '.')? IdentifierPattern '(' ArgumentsPattern ')'

      ConstructorPattern  ⟶  ModifiersPattern (TypePattern '.')?
                              'new' '(' ArgumentsPattern ')'
```

Referenced Java Grammar Elements

The language elements listed here are referenced from the preceding AspectJ grammar rules, but are standard parts of the Java language. When an element's meaning is not obvious, a short explanation is provided.

```
  Modifier  ⟶  'private' | 'public' | 'protected'
             |  'static' | 'abstract' | 'final' | 'strictfp'
```

Arguments arguments to a method or pointcut call

Block a { `block` } of statements

Identifier

Initializer an class instance or static initializer code block

MethodDeclaration

Name a local or qualified name (e.g. `foo` or `org.rura.Foo.foo`)

NameList a comma-separated list of names

NestedClassDeclaration

NestedInterfaceDeclaration

ReturnType

StringLiteral

VariableInitializer expression used to initialize a variable (after the = sign)