Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Generating Pattern-Based Pointcuts for Aspect-Oriented Refactoring Using Inductive Logic Programming

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Mathieu Braem
Promotor: Prof. Dr. Theo D'Hondt
Augustus 2005

Vrije Universiteit Brussel
Faculty of Science
Departement of Computer Science

Generating Pattern-Based Pointcuts for Aspect-Oriented Refactoring Using Inductive Logic Programming

Graduation thesis submitted to obtain a License Degree in Computer Science

Author: Mathieu Braem
Promotor: Prof. Dr. Theo D'Hondt
August 2005

© Vrije Universiteit Brussel, all rights reserved.

Samenvatting

Tijdens het omzetten van object-geörienteerde software naar aspect-geörienteerde software worden we voor het probleem gesteld om goede pointcuts te vinden. Huidige refactorings worden beperkt tot enumeratieve pointcuts. Deze staan het verdere onderhoud van de "-ilities" van een programma in de weg. Ze zijn gekoppeld aan een bepaalde versie van het programma en moeten aangepast worden bij elke verandering van het basis programma. Pointcuts die een patroon beschrijven in de join points vermijden dit probleem. Ze generaliseren de enumeratieve pointcuts door gebruik te maken van gemeenschappellijke informatie over de join points.

In deze thesis stellen we voor een machine learning techniek te gebruiken om pattern-based pointcuts te genereren. Door middel van inductief logisch programmeren brengen we patronen in een verzameling join points aan het licht. Hieruit leren we een pattern-based pointcut. We toetsen deze methode door een "change notification" concern naar een aspect die pattern-based pointcuts gebruikt, te verplaatsen.

Abstract

In the transformation of legacy object-oriented software to aspect-oriented software we are challenged by the problem of finding good pointcuts. Current refactorings are limited to enumeration-based pointcuts, which severely limit the further improvement of the "-ilities" in a program. These pointcuts are too tightly coupled to a specific revision of the base program and may need adaptation on each change to it. Pattern-based pointcuts avoid this problem. They are generalizations of the enumerations, based on the commonalities in the join points.

In this dissertation we propose the use of a machine learning technique to automate the task of generating pattern-based pointcuts. We use inductive logic programming to uncover patterns in a set of join points and to induce a pattern-based pointcut. We validate the technique by refactoring an entangled "change notification" concern into an aspect which uses pattern-based pointcuts.

Acknowledgements

I would like to thank Prof. Dr. Theo D'Hondt for promoting this thesis.

I thank my advisors Kris Gybels and Andy Kellens for the subject of this thesis, and for their guidance and advice while working on the thesis. I also thank ir. Joke Reumers for her help on bugfixing the FOIL implementation.

I thank my family for their support during my studies and the especially stressful period of writing this dissertation.

And finally, thank you friends, for your moral support.

Contents

1 Introduction **1**

1.1 Problem Statement 1
1.2 Proposed Solution 2
1.3 Outline 2

2 Aspect-oriented Programming **3**

2.1 Separation of Concerns 3
2.2 The AOP Approach 4
2.3 Existing AOP systems 4
2.3.1 Composition Filters 5
2.3.2 Hyper/J 6
2.3.3 AspectJ 8
2.4 Summary 11

3 Logic Pointcut Language **12**

3.1 Logic Programming 12
3.2 Logic Programming with SOUL 13
3.2.1 Basic SOUL Constructs 13
3.2.2 Symbiosis with Smalltalk 14
3.3 Logic Meta-programming 15
3.3.1 Meta-programming 15
3.3.2 Logic Meta-programming 15
3.3.3 Logic Meta-programming with SOUL 15
3.3.4 Meta-programming for Java 16
3.4 Logic Pointcut Language 18
3.4.1 CARMA 18
3.4.2 CARMA for Java 18
3.5 Pattern-based Crosscuts 19
3.6 Summary 19

4 Aspect Refactoring **21**

4.1 Object-oriented Refactoring 21
4.1.1 Why Refactoring 21

4.1.2 Bad Smells 22
4.1.3 Refactorings Catalogue 23
4.1.4 Example: *Pull Up Method* Refactoring 24
4.1.5 Tool Support 27
4.2 Aspect-aware Object-oriented Refactorings 27
4.2.1 Example Problem 28
4.2.2 Making Object-oriented Refactorings Aspect-aware 28
4.3 Aspect-Oriented Refactorings 30
4.3.1 Refactorings Catalogue 30
4.3.2 Example: *Method to advice* Refactoring 31
4.4 Problems with Automating Aspect-Oriented Refactorings 35
4.4.1 Flattened Expression Problem 36
4.4.2 Join Point Choice Problem 36
4.5 Summary 37

5 Inductive Logic Programming 38
5.1 Definitions 38
5.1.1 Definition and Terms 38
5.1.2 Example 39
5.2 Properties of Inductive Algorithms 40
5.2.1 Top-down vs. Bottom-up 40
5.2.2 Representation of Background Knowledge 40
5.3 Relative Least General Generalization 41
5.3.1 Definitions 41
5.3.2 Relative Least General Generalization 42
5.3.3 Example 43
5.3.4 Reducing the Size of Clauses 43
5.4 FOIL 44
5.4.1 Algorithm Overview 44
5.4.2 Creating Candidate Literals 45
5.4.3 Selecting the Best Candidate 46
5.5 Summary 47

6 Inducing pattern-based pointcuts 48
6.1 Overall Approach 48
6.2 Induction Algorithm 49
6.2.1 Available Information 49
6.2.2 Algorithms Compared 50
6.3 Example: `stateChanges` Rule 50
6.3.1 Basic Class and Positives 50
6.3.2 Simple `stateChanges` Rule 51
6.3.3 Recursive `stateChanges` Rule 52
6.3.4 Pointcuts 53
6.4 Summary 54

7 Experiments **55**
7.1 Refactor Observer Methods 55
7.1.1 Refactoring with basic information 56
7.1.2 Refactoring with Extra Information 59
7.2 Keywords in methodnames 61
7.2.1 `Address` Class 61
7.2.2 Refactoring the `update` Call 62
7.3 Future work 64
7.4 Conclusion 64

8 Conclusions **66**
8.1 Summary 66
8.2 Conclusion 67
8.3 Technical Contributions 68
8.4 Future Work 68

Bibliography **70**

List of Figures

2.1 AOP system overview 4

4.1 *Pull Up Method* refactoring 25
4.2 Extract method calls refactoring [22] 32

5.1 Visual representation of the hypotheses space 40
5.2 FOIL Algorithm 45

7.1 Background information from basic `Point` class 57

List of Tables

2.1 Composition Filters predefined filters 5
2.2 AspectJ join points 9
2.3 AspectJ primitive pointcut designators 10

3.1 CARMA join points 18

5.1 $Foil_Gain$ values for candidate literals in the first step 47

Listings

2.1 Composition filters example 6
2.2 Hyper/J example 7
2.3 Hyper/J example 8
2.4 AspectJ example aspect 10
2.5 AspectJ example property-based aspect 11
4.1 Code *before* refactoring 26
4.2 Code *after* refactoring 27
4.3 Pointcut for persistent temperatures 28
4.4 Adapted pointcut for persistent temperatures 28
4.5 Account class *before* refactoring 33
4.6 Enumeration based pointcut 34
4.7 Crosscut functionality 34
4.8 Account class *after* refactoring 34
4.9 Simplified pointcut 35
4.10 Flattened expression problem 36
6.1 Base `Counter` class 50
6.2 Basic `stateChanges` rule 52
6.3 Second counter in `Counter` class 52
6.4 `stateChanges` rule if instance variable changed 52
6.5 Extra method calls for `Counter` class 53
6.6 Recursive `stateChanges` rule 53
6.7 `stateChanges` pointcut 54
7.1 Basic `Point` class 55
7.2 Rule from reception join points 58
7.3 Rule from assignment join points 58
7.4 Additions to `Point` class 59
7.5 Rule from reception join points 60
7.6 Rule from assignment join points 60
7.7 `Address` class 61
7.8 Rule from reception join points 63
7.9 Rule from assignment join points 64

Chapter 1

Introduction

1.1 Problem Statement

Current aspect-oriented refactorings produce crosscuts based on enumeration of join points, which hinder further improvement of the "-ilities" in a program; in this thesis we demonstrate how machine learning techniques can learn pattern-based crosscuts.

Aspect-oriented programming is a fairly new programming paradigm. It strives for a better concept of modularity than is possible with present object-oriented languages. Existing object-oriented languages offer modularization by means of classes, methods, packages, *etc*. These, however, do not cope with concerns which occur throughout the source code, but, in fact, have little or nothing to do with their surrounding code. These concerns are said to be *crosscutting*, and cannot be properly localized with the existing techniques. This kind of poor modularization leads to code which is difficult to read, maintain and evolve.

In aspect-oriented software development we separate the crosscutting concerns from the main source and implement these in aspects. An aspect specifies, by means of a pointcut, where in the program it is to be executed. A pointcut is a description of a collection of join points, certain key events in the execution of a program. When such a join point is reached during the execution of the program, the aspect is triggered. As a result, all references to the concern can be removed from the main source and can be contained in their own module. A special tool, an *aspect weaver*, incorporates or "weaves" the concerns into the main source.

We acknowledge the advantages of aspect-oriented programming, and want to use its techniques on existing software. We want to refactor existing object-oriented applications. A refactoring is a behavior-preserving program transformation, with the intention of improving the design, internal structure and reusability of the program. Applying aspect-oriented refactorings means moving crosscutting concerns into aspects. Moving the concerns implies that we have to specify a pointcut that describes the original locations of the removed code. The biggest challenge is to find a good pointcut that expresses the developer's intention, while

maintaining the behavior-preserving property of refactorings.

Automated refactorings are currently limited to enumeration-based pointcuts. These pointcuts are too tightly coupled to a specific revision of the base program. Whenever the program is changed, these pointcuts may need adaptation. Transforming the pointcut into a pattern-based pointcut avoids this problem. It generalizes the pointcut based on the commonalities in the join points. When the base program is changed, the pattern-based pointcut automatically captures new join points that match the described pattern.

1.2 Proposed Solution

We want automated tools able to learn pointcuts that capture the intention of the developer. These pointcuts will not be bound to a specific revision of the base code and even capture new join points that match the pattern, as they are appear by changing the program.

We use an algorithm from the field of machine learning. More specifically, we apply inductive logic programming to our problem. ILP is a technique that allows the creation of first-order logic rules from a set of examples and background information. We map the join points we want to describe to the positive examples for the algorithm and use the rest of the program as background information. The result of this algorithm is a pattern-based pointcut.

1.3 Outline

In this dissertation we show how we map the problem of find an intensional pointcut to using inductive logic programming to learn a logic rule. The document is organized as follows: chapter 2 gives an overview of aspect-oriented development. We show how aspects offer better separation of concerns and discuss three different approaches to AOP. Chapter 3 gives an introduction to logic programming and the logic language SOUL. We discuss logic meta-programming and show how it provides the building blocks for a logic pointcut language CARMA. We adapted CARMA for use on Java source code. Chapter 4 gives an overview of refactoring software. We take a look at object-oriented refactorings and how to make these aspect-aware. We also discuss aspect-oriented refactorings and what problems arise in automating these refactorings. Chapter 5 discusses a machine learning technique, named inductive logic programming. We give an overview of its concepts and show two example algorithms. In chapter 6 we apply one such ILP algorithm, FOIL, to learn pattern-based pointcuts. We show how well this maps to the goal of ILP, namely inducing first-order logic rules. In chapter 7 we validate this claim and show how we induce a number of pointcuts in real examples. We finish the dissertation in chapter 8, where we draw our conclusions from the experience gained during this research. We take a look at future work to improve and continue this research.

Chapter 2

Aspect-oriented Programming

This chapter introduces aspect-oriented programming. We show how aspect-oriented programming simplifies the tasks of developing and maintaining software. We list three different approaches to aspect-oriented programming.

2.1 Separation of Concerns

Developing large software applications is a difficult task. The size and complexity of the task make it necessary to develop some techniques that allow the developer to complete his job with more ease. Trying to tackle a large application as a whole is nearly impossible. We have to decompose the program into smaller parts that are manageable. Afterwards we integrate these parts and obtain our large program. Before the smaller parts can be implemented, we have to carefully choose on which criteria we base our modularization.

When programming the application, a large number of requirements are to be met. Good decompositions let the separate parts address only a specific concern. They each implement a specific subset of the requirements. Being able to handle the concerns separately greatly enhances programming and maintaining software applications. Developers can focus on a single aspect of the program, able to ignore others. Dijkstra [9] named this idea "separation of concerns". Ideally, all modules should handle only a single concern and concerns should be programming in a single module.

Unfortunately, it is not possible to separate all concerns with a single decomposition. Some concerns are hard to introduce in the program into a single module. Adding, for example, concurrency to the system, means implementing a synchronization object, and updating the methods that need synchronization to include support for it.

It is due to this fact that in current systems, many concerns are scattered across modules. They become entangled with other modules and the software becomes harder to understand and maintain. Being able to separate these crosscutting concerns at the implementation level, as we can on the design level, brings a higher

level of abstraction to the development process. Aspect-oriented programming [19] is a technique to modularize these concerns as well, removing code scattering and entanglement.

2.2 The AOP Approach

Different AOP mechanisms exist. We can distinguish between domain-specific and general-purpose AOP systems. Domain-specific languages serve only a single purpose. The COOL language, for instance, intents to provide better support for synchronization and guard handling between Java threads. The general-purpose systems are not limited to a single goal. In this dissertation we focus on the general-purpose systems.

In aspect-oriented languages, the programmer describes all concerns separately from the base code and combines them to their final form using some kind of tool support. We specify these aspects by means of pointcut and advice. A pointcut is used to express where to execute the concerns actions. A pointcut is a description of a set of events in the source code. These events, named *join points*, are well defined points in the program execution, e.g. method calls or assignments. When the join points described in the pointcut are reached during the program execution, the advice part of the aspect is executed. This way a concern can be expressed in a single location, cleanly separated from other concerns definitions. It can express what actions to need to be done, and specify at what time they are to be executed. Figure 2.1 shows how the different components in an AOP system work together. The figure shows a base program and the separate definition of an aspect. After applying a tool, called an aspect weaver, we obtain the final program.

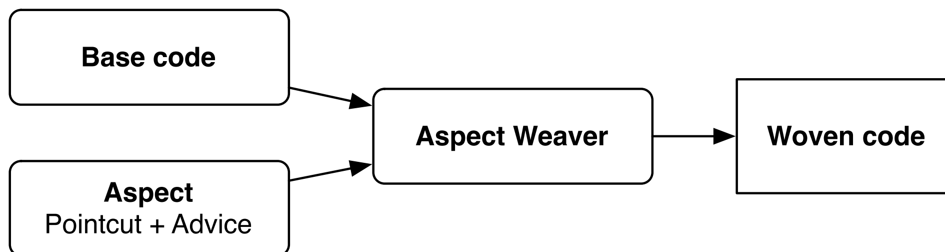

Figure 2.1: AOP system overview

2.3 Existing AOP systems

In this section we take a closer look at three different approaches to aspect-oriented programming. First we discuss Composition Filters, next we look at Hyper/J, and finally we discuss AspectJ.

2.3.1 Composition Filters

The first approach we discuss is Composition Filters (CF) [6]. CF primarily focusses on software evolution. Bergmans and Aksit [6] see two possibilities to extend existing software: either change the definition of already existing classes or extend them. Using the traditional object-oriented mechanisms, extending classes may either take an inheritance-based approach or can use aggregation. Inheritance will override the methods that need to be changed. All messages are implicitely understood by means of the standard OO inheritance rules. With aggregation, all messages understood by the base class need to be implemented and delegated. The messages that need to be changed are implemented differently and can still be optionally forwarded to the delegate. However, in either case of extending the classes, the names of the classes will have to be changed in the programs source before the new, extended classes can be used. Furthermore, changing only a single feature in a class, can lead to numerous new classes and method implementations.

The idea behind composition filters is to capture the messages sent to objects. It is an extension to the model of OO programming. Before a message is sent or received, it is subjected to an array of filters. Each method can have several filters of varying types attached to it. They are stored in an ordered collection and are executed in order. Messages pass each filter in the set until it is discarded or can be dispatched, either executed by a local method or delegated to another object. Composition filters offer an orthogonal extension, which means that the filters do not need to know of each other.

The filters define the behavior of an object. The filters can inspect and manipulate the messages that are sent and received by an object. In the implementation of a filter, two types of methods can be defined: regular methods and condition methods. The regular methods implement the functional behavior of the object. The conditions support the filters to decide how to manipulate messages.

Each filter will either accept or reject a message, the precise semantics depending on the type of the filter. Table 2.1 lists some of the predefined filter types.

Dispatch The message is dispatched to the current target object if the filter accepts the message.

Error An exception is raised if the filter rejects the message.

Wait Until the evaluation of the filter expression is accepted, the message is queued. When it is accepted, the message continues to the next filter.

Meta If the message is accepted, it is reified as a parameter to another (meta-) message. The meta message can inspect and manipulate the message and re-activate it for execution.

Table 2.1: Composition Filters predefined filters

We take a look at a small example in listing 2.1. We want to implement a counter for the total number of floors an elevator has moved. We call this the “odometer” concern and we want to apply it to an existing `Elevator` object. We declare a filter of type Meta with the name `odometer`. The filter consists of a single filter element. Before the `climb` and `descend` messages are evaluated, the `CountFloors` message has to be evaluated. This message will count the number of floors moved and re-activate the execution of the first message.

```
1 odometer : Meta = {
2   countFloors => {climb, descend},
3   ... // other filters
4 };
```

Listing 2.1: Composition filters example

2.3.2 Hyper/J

Hyper/J is a Java implementation of hyperspaces [27, 30], the second AOP approach we discuss. The principle of separation of concerns tells us to decompose software in modules, each dealing with a particular area of interest, called concerns. These concerns can be data types or classes, features, roles, *etc*. In OO programming, we can only decompose using classes. In functional programming, we can decompose by functions and in rule-based systems by rules. This problem is named the “tyranny of the dominant decomposition” [31]. Developers have to choose between decomposition dimensions early in the development process and are bound to it.

The proposed solution to this problem is “multi-dimensional separation of concerns” [31], meaning the separation of concerns involving multiple, arbitrary dimensions of concern, all of equal importance and occurring simultaneously. Furthermore it should be possible to handle new concerns and new dimensions of concern dynamically, as they arise during the software lifecycle. Ideally, the dimensions of concern are orthogonal, but achieving this kind of independency is rarely possible in practice. Overlapping and interacting concerns need some points of interaction to maintain the appropriate relationships.

Separation of concerns encompasses three distinct components: the identification of concerns, the encapsulation of these concern, and last, the integration of the concerns. The hyperspaces approach supports these three components by means of concern spaces, hyperslices and hypermodules.

A concern space holds all software units in a program, i.e. the syntactic constructs in a language. They can be primitive such as methods and instance variables, or they can be compound units, such as classes or packages. It is the concern spaces task to organize the units in the software so as to separate all important concerns, how they relate to each other and how they can be integrated in to software systems.

A hyperspace is a specially structured concern space to support multi-dimensional separation of concerns. Its units are organized in a multi-dimensional matrix, of which each axis represents a dimension of concern. Each dimension can thus be viewed as one particular software decomposition.

Sets of units in the concern matrix cannot be treated as separate modules. Hyperslices allow the grouping of concerns into implementation modules. Each of the hyperslices is declarative complete, which means that they must declare everything to which they refer. By doing so, they are not tightly coupled to other hyperslices and remain self-contained. Hyperslices can be combined if the required declarations are fully defined. This tolerates flexible configuration and reuse of hyperslices.

Finally, hypermodules consists of a set of hyperslices being integrated and a set of relationships, which specify how the hyperslice relate and should be integrated. A hypermodule, too, is a declarative complete module, and can be used as if it was a hyperslice.

Hyper/J is an implementation of the hyperspaces approach. The supported units are packages, interfaces, classes and members. Combining sets of these units in hyperslices, and integrating them into hypermodules results in Java class files for all hypermodules produced. The classes can be used for further development.

The following example in listing 2.2 demonstrates the addition of a "odometer"-concern to an elevator class. We first define a new dimension: the `Feature` dimension. The first mapping indicates that all of the units in the package `demo` address the `Kernel` concern. The `move` operation and the `count` operation address the `Movement` and `Odometer` concerns, respectively. The `climb` and `descend` operations are operations that will be combined from separate concerns. Because the Java software units are methods, they have to be excluded for now. We will use composition rules to invoke these features.

```
1 package   demo     : Feature.Kernel
2 operation move     : Feature.Movement
3 operation count    : Feature.Odometer
4 operation climb    : Feature.None
5 operation descend  : Feature.None
```

Listing 2.2: Hyper/J example

The hypermodule in listing 2.3 combines the hyperslices from the previous listing. The "mergeByName" indicates that operations with the same name are to be combined. The "equate operation" relationship finally completes the previously undefined declarations for the `climb` and `descend` methods. They are a combination of the `move` and `count` operations from the `Movement` and `Odometer` features.

The result of composing the hyperslices into this hypermodule is that all operations are defined, and the "odometer"-concern is added to the original elevator behavior.

```
1  hypermodule Elevator_with_odometer
2    hyperslices:
3      Feature.Kernel,
4      Feature.Movement;
5      Feature.Odometer;
6    relationships:
7      mergeByName;
8      equate operation Feature.Kernel.climb,
9                       Feature.Kernel.move,
10                      Feature.Odometer.count;
11     equate operation Feature.Kernel.descend,
12                      Feature.Kernel.move,
13                      Feature.Odometer.count;
14 end hypermodule;
```

Listing 2.3: Hyper/J example

2.3.3 AspectJ

AspectJ [1, 20, 21] is an extension to Java. It is a general-purpose aspect-oriented programming language. In this section we discuss the core features of AspectJ. We start with an overview of the join point model. We continue by showing how to identify join points in program code, followed by how to affect the implementation at join points. Next we take a closer look on property-based crosscuts. We conclude this section with the discussion of two examples.

AspectJ supports two types of crosscutting concerns. The first allows us to define additional implementation to run at certain points in the program. This is called dynamic crosscutting and is the main focus of our discussion. The other type of crosscutting allows the programmer to define new operations on existing types. This type is named static crosscutting and is commonly referred to as *open classes* or *introductions*. Existing classes can be extended with new fields and methods. Also, the class hierarchy of the program can be altered.

The purpose of the join point model is to allow separation of concerns. The base code of a program can remain oblivious of the aspects' influence on the execution of the code. Join points are well defined points in the execution of the program. We can relate them to the nodes in a runtime object call graph. The nodes include points where an objects receives a method call and points where an objects field is referenced. Edges are control flow relations between the nodes. AspectJ can address different kinds of join points. An overview of these kinds is given in table 2.2.

We use pointcut designators to identify join points in program code. A pointcut is simply a description of a set of join points and the context of those join points. Pointcut designators are used to match certain join points at runtime. Next to the primitive pointcut designators for the different types of join points (from table 2.3),

Join point	Description
message send constructor invocation	An object sends a message to another object. The join point is associated with the first object.
message reception construction invocation reception	An object receives a message. Here the join point is associated with the receiving object and occurs before control has been passed to the any methods or constructors.
method execution constructor execution	Upon execution of a method or constructor.
field get	A field of an object or class is read.
field set	A field of an object or class is changed.
exception handler execution	Upon invoking a method handler.
class initialization	A field of a class is being initialized.
object initialization	A field of an object is being initialized during object creation.

Table 2.2: AspectJ join points

AspectJ also includes primitive pointcut designators which match on certain properties of pointcuts. Table 2.3 lists and describes primitive pointcuts designators that may be used. Pointcut designators can be combined using “and”, “or” and “not” operators. Also, pointcuts can be named to create user-defined pointcut designators. These can be used wherever other pointcut designators appear.

AspectJ uses a method-like mechanism to declare certain code to be executed at each of the join points captured in the pointcut. The advice that should be executed is written as plain Java code. AspectJ allows before, after and around advice, of which the first two simply add to the computation at the join point by executing the advice code before or after the join point. Around advice replaces the computation at the join point. If needed, the `proceed` call defers execution to the original join point.

Next to the enumeration-based pointcuts, AspectJ also supports descriptive, property-based pointcuts. The pointcuts we have seen so far, are all defined by means of explicitly enumerating method signatures. Property-based crosscutting allows to define a pointcut by referring to certain other properties of join points. AspectJ features wildcards in pointcut designators and control-flow based pointcut designators. Wildcards can replace any string or substring in the method signature of the pointcut designators. AspectJ will find all types or names that match the given wildcard expression. For example, the method signature “`* User.get*()`” expresses all zero-argument methods in the `User` class that start with “get” and return any type.

AspectJ includes primitive pointcut designators by which join points can be picked based on whether or not they occur in a control-flow relationshow with

Pointcut designator	Description
call(signature) execution(signature)	Matches call or execution join points at which the method signatures matches.
get(signature) set(signature)	Matches field get and set join points at which the signature matches.
handler(TypeName)	Matches the exception handler execution joinpoints for which the type of the thrown object matches.
staticinitialization(TypeName) initialization(TypeName)	Matches class or object initializations of the given type.
within(ClassName) withincode(signature) cflow(pointcut)	All join points where the associated code is defined in the specified class. All join points where the associated code is defined in the specified method. Matches any join point in the control flow of the specified call. This matches all join points after the call and before the execution is finished.

Table 2.3: AspectJ primitive pointcut designators

other join points. These take other pointcut designators as arguments. Discussion on these pointcuts can be found in table 2.3.

Finally, aspects in AspectJ are very similar to Java classes. In those aspects, pointcut designators and advice are defined. Aspects can even have methods and instance variables. Similar to subclassing in OO, AspectJ allows a sub-aspect to inherit from another aspect. The sub-aspects can extend the definition of the base aspect. This introduces the possibility to create an aspect library in which sub-aspects parametrize the behavior.

In listing 2.4 we give an introductory example of an aspect in AspectJ. In the example we define an aspect which keeps track of the total number of floors an elevator object travelled so far.

```
1  aspect Odometer {
2    static int floor = 0;
3    static int distance = 0;
4
5    pointcut moves(int destination):
6        (call(void Elevator.climb(int)) && args(destination)) ||
7        (call(void Elevator.descend(int)) && args(destination));
8
9    after(int destination): moves(int) && args(destination) {
10     distance +=  Math.abs(floor - destination);
11     floor = destination;
12   }
```

```
13 }
```

Listing 2.4: AspectJ example aspect

We assume an `Elevator` class is defined and has the methods `climb` and `descend` defined. We define a pointcut (lines 5 to 7) which should match all join points where one of these two methods is called. It binds the argument of the method call to the `destination` parameter. Next, on line 9, we instruct the aspect weaver to inject the advice after each join point covered by our `moves` pointcut. The advice adds the number of floors moved to the total distance travelled (line 10), and keeps those numbers in instance variables of the aspect.

The second example in listing 2.5 demonstrates property-based crosscutting. We define a pointcut (line 2) that captures all join points that occur after the `climb` and `descend` methods are called on an `Elevator` object, but before the method has finished. The advice on line 6 can now do some testing before execution at those join points continues.

```
1  aspect InternalMaintainance {
2    pointcut monitored(Elevator e):
3      cflow(calls(* Elevator.climb(int))) && target(e) ||
4      cflow(calls(* Elevator.descend(int))) && target(e);
5
6    arround(Elevator e): monitored(Elevator) && target(e) {
7      if (...) { // internal testing
8        proceed; // ...
9      }
10   }
11 }
```

Listing 2.5: AspectJ example property-based aspect

2.4 Summary

This chapter introduced aspect-oriented programming. A good decomposition of a software system puts each concern in a separate module. We identified the problems in object-oriented programming languages and showed how aspect-oriented languages can help in achieving a better separation of concern at the implementation level. We have shown a few different AOP approaches and how they manipulate the join point structure of the base program.

Chapter 3

Logic Pointcut Language

In this chapter we discuss the logic programming paradigm and logic meta-programming. We show a pointcut language based on logic meta-programming and discuss pattern-based pointcuts.

3.1 Logic Programming

In logic or declarative programming, we are concerned with what needs to be accomplished. We express our program as a series of facts and queries over these facts. The computational part is defined in the logic inference engine.

In this section we discuss the logic programming paradigm. We will compare it to other well known programming paradigms and take a closer look at SOUL.

Imperative programming The emphasis with imperative programming is on the state maintained in programs. The program specifies, in a number of steps, how to manipulate the state of the program. Programs written in procedural or object-oriented languages are generally written in imperative style. Typical examples are C, C++, Pascal, Java.

Functional programming In this paradigm computation is performed by transforming entities into new entities, just as mathematical functions would. A program is a large transformation, composed of a number of smaller ones. Examples include LISP and Scheme (disregard the destructive operations) and Haskell.

Logic Programming In logic programming programs are specified by declaring the base knowledge of the problem field and the relationships between them. These are the facts of the program. The computational part exists of rules that specify how to derive new information from these facts. By posing a query, the system will try to prove it to be correct or will try to prove it to be incorrect. This system is based on first order logic. Examples of these languages are Prolog and SOUL.

We can now clearly distinguish programming paradigms. In the next section we take a closer look on a logic programming language, SOUL.

3.2 Logic Programming with SOUL

The Smalltalk Open Unification Language (SOUL) is derived from Prolog and was developed by Roel Wuyts at PROG [36]. SOUL is implemented in Smalltalk.

3.2.1 Basic SOUL Constructs

We show the syntax of SOUL by some discussing some basic examples. These should not be difficult to follow. Readers already familiar with Prolog will notice the following differences: variables are prepended with a question mark (`?`) and need not be capitalized, `if` replaces `:-` when writing a rule and lists are surrounded by angular brackets (`<...>`) instead of brackets (`[...]`).

Writing a program in SOUL consists of providing the interpreter with rules and facts to reason with.

```
parent(jim, bob).
parent(bob, julie).
parent(bob, eric).

grandParent(?x, ?y) if
  parent(?x, ?z),
  parent(?z, ?y).
```

In the first three lines of this example we express facts that we know, basic knowledge. In the last rule we form a rule that defines the grandparent relationship. We can now query the interpreter. In the following example we query all grandparent relationships.

```
if grandparent(?x, ?y).

{?x → jim, ?y  → julie}
{?x → jim, ?y  → eric}
```

The results are variable bindings for every possible solution. The interpreter correctly bound the variables to names and paired them correctly for the grandparent relationship. Note that with the two-way binding property of the variables, we can also query to verify, i.e. the following example would result in true.

```
if grandparent(jim, eric).
```

In the following example we take a closer look at the list structure, a native data structure in the logic language. Lists are denoted between angular brackets (< and >). We can bind the head and the tail part of the list seperately using the bar notation. In <?X | ?Y>, ?X is bound to the head of the list and ?Y is bound to the tail part. This example implements a predicate to append two lists.

```
append(<>, ?Y, ?Y).
append(<?X | ?Xs>, ?Y, <?X | ?Z>) if
  append(?Xs, ?Y, ?Z).
```

This little program has two rules to describe the predicate. In order to resolve a query, SOUL will apply each rule in order of specification. The first line describes the base case, appending an empty list to another list ?Y is the list ?Y again. The second rule recursively defines the append. Take each element from the first list and append it to the other list. Applying this rule recursively results in an ever shrinking first list, of which each element is added to the second list. At a certain point, the first list will be the empty list and SOUL will apply the first rule.

The list is actually a special kind of functor. Functors are like normal predicates, but are never evaluated. They only have structural meaning. The list data structure is the successive application of the dot functor with some syntactic sugar. The following equation holds: <x, y, z> = .(x,.(y,.(z,.()))).

3.2.2 Symbiosis with Smalltalk

SOUL was designed with language symbiosis with Smalltalk in mind.

- Smalltalk terms are denoted between brackets and can contain logic variables. We can let Smalltalk values live in logic variables and use the variables in parts of Smalltalk code. The following example takes a Smalltalk value in a SOUL variable and will bind its size to another variable.

```
size(?collection, ?size) if
  equals(?size, [?collection size]).
```

- Smalltalk clauses can be used instead of logic clauses, as a result they must return in a boolean value. In the following example, the greaterThan predicate compares two Smalltalk values. Smalltalk clauses can be used for their side effects, e.g. writing something to the screen. This is also demonstrated in the example. Note the explicit return of the truth value.

```
greaterThan(?x, ?y) if
  [?x > ?y].

writeString(?string) if
  [Transcript show: ?string. true].
```

- Quoted terms are just as strings, with the exception that they too can contain logic variables. These variables will be substituted upon evaluation. Quoted terms are delimited by braces. The following example demonstrates a great use of quoted terms: code generation. In this case, the string is a Smalltalk expression. It will not be evaluated, but the logic variables will be substituted.

```
generateShowClassName(?class, ?code) if
  equals(?code,
        {Transcript show: ?class asString}).
```

3.3 Logic Meta-programming

3.3.1 Meta-programming

Every program is a computational system which operates on its problem domain. Entities from this domain are modeled, and its representations are used in the systems computation. A particular class of systems consists of those systems that have computational systems as their problem domain. These computational systems are called meta systems. A program of this system is a meta program.

We can also distinguish programming languages based on the problem domain they act on. On one hand, we have domain specific languages, specifically designed to deal with a particular problem domain. E.g. Postscript is a domain specific language for instructing printers. On the other hand, we have general-purpose languages. These are not designed to cope with a specific problem domain's needs, e.g. Smalltalk and Java.

We notice the same distinction with meta-programming languages. These languages are, generally speaking, designed to operate on a specific programming language. This language of focus is named the base language, while we refer to the meta-programming language as the meta language.

3.3.2 Logic Meta-programming

With the definitions for logic programming and meta programming from the previous sections we define logic meta programming (LMP): It is the use of a logic programming language at the meta level to manipulate programs built in some underlying base language.

We can generally place the applications of LMP in five categories: verifying source code, extracting information from source code, transforming source code, generating source code, aspect-oriented programming [17].

3.3.3 Logic Meta-programming with SOUL

As already discussed in a previous section, SOUL supports a number of special contructs to interact with Smalltalk. This section shows how a special library for

SOUL supports logic meta-programming.

To support access to Smalltalk entities, SOUL uses a meta language interface (MLI). This interface bridges Smalltalk entities and their SOUL representations. Whenever the SOUL interpreter is in need of a Smalltalk element, the MLI is called and returns the SOUL representation of the element. The entities will have a logic representation using functors. The most basic elements, e.g. classes, use a this simple representation. More complicated elements, e.g. message send statements, will be represented by their parse tree.

On top of this MLI a large library of predicates is built. LiCoR (Library for Code Reasoning) is designed as a set of layers. Predicates from a layer only use predicates defined in that layer, or predicates from the layers below it. In the following overview we take a closer look at these layers, starting with the most basic one.

- The logic layer contains the most basic predicates, needed for basic logic programming. These include predicates for handling lists, arithmetic and logic meta predicates (`var` for variable checking, `assert` and `remove` for adding and deleting from the logic database, *etc*).
- Predicates from the representational layer define the logic, structured representation of base programs. These include predicates for representing classes and their hierarchy, methods, *etc*.
- The basic layer holds auxiliary predicates aiding in reasoning about programs. Predicates from this layer provide higher lever information, given what can be extracted using the predicates from the representational layer.
- In the design layer, the highest level predicates reside. They reason about the use of programming conventions, design patterns, *etc*.

3.3.4 Meta-programming for Java

This dissertation focusses on Java source code. We discuss two meta-programming systems for Java in greater detail.

TyRuBa

TyRuBa was implemented to facilitate in type-oriented logic meta-programming [7, 8]. The name is an acronym and stands for Type Rule Base. It is a logic programming language, which provides some facilities for manipulating Java source and enables code generation.

TyRuBA can represent Java programs using predicates. Predicates exist for classes, instance variables, methods, *etc*.

TyRuBa has the notion of quoted code blocks, they are comparable to SOUL's quoted terms. Quoted code blocks can be used to represent Java code in a string-like representation. The quoted code blocks can contain variables and compounded

terms, which will be substituted upon evaluation. This makes TyRuBa very suited for template based meta-programming. The following example uses code generation to add a single method to each known class. Note that this example is simplified for readability purposes, TyRuBa's syntax differs from SOUL's and Prolog's, but should still be easily readable.

```
method(?class, printClassName,
       { public void printClassName() },
       { System.out.println("?class"); }) :-
  class(?class).
```

In this case, the method predicate takes four arguments. The first argument is the class where the method is defined. Next is the method name, followed by the signature, denoted as a quoted code block. The last argument is the body for the method, again a quoted code block. The variable `?class` will be bound to every class in the system and will be replaced in the string that forms the body of the method.

Using these kind of rules to transform an existing representation in of a Java program into another by adding or retracting facts, results in a transformation of the program itself.

Irish (SoulJava)

Irish is an extension to SOUL for Java [3]. Irish has an external dependency on Frost [2] to parse java code. Frost imports Java source into a Smalltalk image by parsing the source code and instantiating Smalltalk objects, representing a parse tree of the source. The parse tree can then be manipulated, and printed out as Java code.

Irish provides the glue between these objects and the SOUL evaluator. It extends the meta language interface (MLI) and adapts LiCoR to work on the currently parsed Java code. With LiCoR able to access the information about the Java code, logic meta programming on this code is possible from SOUL.

For example, reading a Java source file with Irish will store it into a *JavaCodeRepository*. Many of the predicates available to reason about Smalltalk code are available for Irish. The `class(?c)` predicate will query all classes, while `method(?m)` queries all methods. Additional predicates are also available. The following example queries the interfaces a class `ClassA` implements.

```
classWithName(?class, ClassA),
classInterface(?class, ?interface)
```

Join point	Description
message send	An object sends a message to another object. The join point is associated with the first object.
message reception	An object receives a message. Here the join point is associated with the receiving object and occurs before control has been passed to the any methods or constructors.
assignment	The state of an object is updated by executing an assignment statement.
reference	The state of an object is inspected by executing a reference statement.
block execution	A Smalltalk block is executed.

Table 3.1: CARMA join points

3.4 Logic Pointcut Language

3.4.1 CARMA

CARMA, previously named Andrew [14], is a logic pointcut language for Smalltalk. Its join point model, like AspectJ's, is based on the key events happening in object-oriented programs. Namely sending and receiving of messages, and inspecting and changing the state of objects. Table 3.1 lists the types of joinpoints available in CARMA.

Pointcuts are defined as logic queries about join points. The following example will bind all reception join points to the `?jp` variable, restricting the matching join points to those that are called without arguments to the message (specified with the empty list <> for the `?arguments` variable). For every matching join point, the `?selector` variable will be unified with the selector of the message at the reception join point.

```
if reception(?jp, ?selector, <>).
```

3.4.2 CARMA for Java

For the experiments in this thesis, we defined a pointcut language for Java, based on CARMA. We used a join point model similar to CARMA's, offering the same kinds of join points to crosscut. We can also use some extra information available in Java programs, e.g. information about packages, types, interfaces, *etc*.

The following example demonstrates this by defining a pointcut that captures all reception join points of methods, of classes in the `Security` package, that contain an assignment statement. The aspect defined on the last three lines instructs the weaver to insert a print instruction before each monitored method.

```
monitored(?jp) if
  classInPackage(?class, Security),
  methodInClass(?method, ?class),
  reception(?jp, ?method),
  inMethod(?statement, ?method),
  isAssign(?statement, ?varName, ?value)
```

3.5 Pattern-based Crosscuts

The advantage of a logic pointcut language is that more complex pointcuts can be defined. The unification system of the logic language is more powerful than the simple wildcard mechanism, as present in AspectJ. The asterisk that is available in the wildcard mechanism is a lot like the anonymous variable in logic programming. Being anonymous, we cannot use the bound value later in the pointcut. Named variables on the other hand, allow us to use this value, by unifying the variable with other variables or values. Based on the basic predicates of CARMA and SOUL's declarative framework, logic rules can express arbitrary complex pointcuts.

It is important to correctly describe the join points in a pointcut. A pattern-based pointcut exploits the "patterns" or commonalities in the join points that are to be crosscut. This decouples the pointcut from a particular version of the base program and the pointcut will automatically capture new join points that match this pattern when the base program is changed.

The following example implements a rule to find those assignment join points which are the execution of an assignment statement in an instance initialization method. We depend on the `instanceCreationMessage` predicate from SOUL's declarative framework.

```
initialization(?jp, ?class, ?varName, ?initVal) if
  class(?class),
  instVar(?class, ?varName),
  assignment(?jp, ?varName, ?preInitVal, ?initVal),
  within(?jp, ?class, ?selector),
  instanceCreationMessage(?class, ?selector).
```

3.6 Summary

This chapter showed the basic principles behind logic programming. We used the language SOUL to discuss logic meta-programming on Smalltalk and Java programs. We also studied how the use of logic meta-programming results in a logic pointcut language. We showed how powerful pattern-based pointcuts can be expressed in a logic pointcut language.

In a later chapter we discuss inductive logic programming. A machine learning technique to induce logic rules from a number of facts. We will map this technique

to uncover pattern-based pointcuts in existing programs.

Chapter 4

Aspect Refactoring

In this chapter we will discuss refactoring of programs. We first give an overview of the topic and show a number of conventional object-oriented refactorings. We show how to apply refactoring to aspect-oriented programs and the discuss the problems that show up.

4.1 Object-oriented Refactoring

Refactoring is the process of altering source code in order to improve its design, internal structure and reusability, without changing any behavior. Refactoring aims to change existing code in a disciplined manner, while minimizing the risk of introducing bugs, and to clean up the code or change its design. Refactoring improves the design of software and makes it easier to understand. It helps in finding bugs in the code and aids in maintaining a constant speed in software development.

4.1.1 Why Refactoring

The goal of software engineering is to create working and reusable software. Ideally the software system is built from scratch, with a complete understanding of the problem domain. In practice, however, software engineering often requires us to change, add or remove functionality in an existing piece of software. When these systems are not well designed, lacking in modularity and reusability, this can prove to be a difficult task. Refactoring the code will remove these bad properties and prepare the code for easier addition and removal of components and functionality.

Refactoring improves the structure of a program. During a program's lifetime, code quality decays. Changes made to realize short-term goals or without full comprehension of the design make the code lose structure. An ill-designed program makes it very hard for the engineer to cleanly change the program. This is the cumulative effect of code decay. The harder it is to grasp the design of code, the harder it will be to maintain this design, and the more rapidly it will decay.

While reading an application's source, assumptions need to be made about the code. The more often these need to be made, or the more complex the assumptions are, the more likely it is to miss bugs in the code. Refactoring leads to proper design and makes the code easier to read and understand. The code better communicates its purpose to the reader. The number and complexity of the assumptions made by the reader will be reduced, and will help in spotting bugs or problems in the code.

Deciding to step back from the normal programming process and refactor the code can happen at almost any time during the software lifecycle. The most convenient times to refactor are when adding new functionality to some software. If adding a feature would be easier with a change of design, it is time to refactor. The process of transforming the code improves the understanding of the code and aids in localizing and, ultimately, fixing of bugs. In the same spirit, code reviews are a good time to refactor. When reviewing the code, the engineer considers the need for refactoring and if it would help in improving the structure of the code.

An important property of the refactoring process is that it should be behavior-preserving.

4.1.2 Bad Smells

There are certain indicators in a program's code that suggest the possibility of refactoring. They are patterns that occur frequently and generally indicate bad design. We can resolve these problems by using a fixed pattern of refactorings to obtain a better design. In this section we discuss a number of these common indicators that reveal bad design, named bad smells.

- **Duplicated code** If the same code structure occurs in more than one place in a program's source, the design will most certainly improve by unifying them. Removing duplicated code results in less code to understand and makes it easier to modify it correctly.

- **Long method** Having large chunks of code makes it difficult to completely grasp its intention. The small overhead of switching context when reading the subprocedure calls can be avoided by properly naming these procedures.

- **Large class** A class that is trying to do too much is often spotted by the large number of instance variables. As is a class with too much instance variables, so is a class with too much code an indication for duplicated code.

- **Long parameter list** Long parameter lists are hard to understand, as they become inconsistent and difficult to use. They also tend to change as more data is needed. Object-oriented programs can have smaller parameter lists than traditional programs because the objects can be asked to get the data needed. A method needs only as many parameters as required to get to all the data needed by asking the objects to get it for them, i.e. an object passed as a parameter can be queried to retrieve further arguments for the method.

The goal of identifying and naming these bad smells is to make the task of finding and fixing them easier. We can apply a number of different refactoring techniques to transform the code. The next section discusses these common refactoring techniques.

4.1.3 Refactorings Catalogue

A *refactoring* is an agreed upon technique to transform a code fragment along a certain pattern. There exist a number of these refactorings, which are bundled in catalogues [12]. A refactoring can be defined using a standard format. The standard form includes a name for the refactoring, a summary which outlines in what situations the refactoring is appropriate and what it does. The motivation describes why the refactoring should be done and in which circumstances it shouldn't be done (the pre- and postconditions). A step-by-step guide of how to carry out the refactoring is detailed in the mechanics section.

The refactorings can be catagorized by their goal. A large part of the refactorings deal with *composing methods*, the problem of packaging code correctly. Long methods contain too much information, which often gets buried by complex logic. Once a method is correctly broken down into smaller parts, it becomes easier to understand the parts. A fundamental decision in OO design is deciding where to put responsibilities. *Moving features between objects* resolves the problem of incorrect design by moving the behavior around. *Organizing data* refactorings make working with data easier by defining new types that hide the data and lets the programmer deal with objects with behavior. A number of refactorings deal with *simplifying conditional expressions*. Good interfaces for objects can be obtained by *making method calls simpler*. Good interfaces require proper names for methods and parameters lists that are easy to remember. Finally, *generalization* deals with inheritance and delegation related refactorings.

This sections summarizes a number of the more common refactorings, without going in too much detail. More refactorings can be found in Fowler et al. [12]. The following section illustrates the refactoring process in more detail for a single example.

- **Extract method** When a method contains a code fragment that can be grouped together, the fragment should be turned into a method whose name explains the purpose of the method. The key is to keep the semantic distance between the method name and the body as small as possible. Extracting methods should improve the clarity of the code, and this will certainly benefit from well-named methods.

- **Move method** A method is, or will be, using or used by more features of another class than the class on which it is defined. Refactor this by creating a new method with a similar body in the class it uses most. The old method should be turned into a simple delegation, or completely removed. Moving

methods lightens classes with too much behavior. It will also remove the tight coupling from classes that are collaborating too much.

- **Rename method** Methods should be named in a way that clearly communicates their intention. When a method does not reveal its purpose, its name should be changed. Good naming will make the development proces much easier.

4.1.4 Example: *Pull Up Method* Refactoring

In this section we discuss the "pull up method" refactoring [12]. This refactoring is applicable when methods with identical results are defined on subclasses. To resolve this, we will move these definitions to the superclass. Figure 4.1 illustrates the concept.

Motivation

It is important to eliminate duplicate behavior. Two duplicate methods may seem fine, but there is a risk that a change to one will not be made to the other. The "pull up method" refactoring will often come up after other steps. It could be possible that a certain refactoring of methods in two classes make them end up in essentially the same method. In this case it is easy to generalize them afterwards. The refactoring can also occur when a subclass method overrides a superclass method, but essentially does the same thing. It can also happen that the body of the method refers to features of the subclass that are not available on the superclass. In this case, the other method should be generalized as well, or an abstract method should be made in the superclass.

Mechanics

These are the steps that need to be taken to apply the "pull up method" refactoring.

1. Inspect the methods to ensure they are identical. If they look like they do the same thing, but are not identical, the "algorithm substitution" refactoring [12] can be applied to one of the methods to make them identical.

2. If the methods have different signatures, change the signatures to the one needed in the superclass.

3. Create a new method in the superclass and copy the body of one of the methods to it.

4. Delete all subclass methods such that only the superclass method remains.

5. Take a look at the callers of this method to see whether a required type can be changed to the superclass.

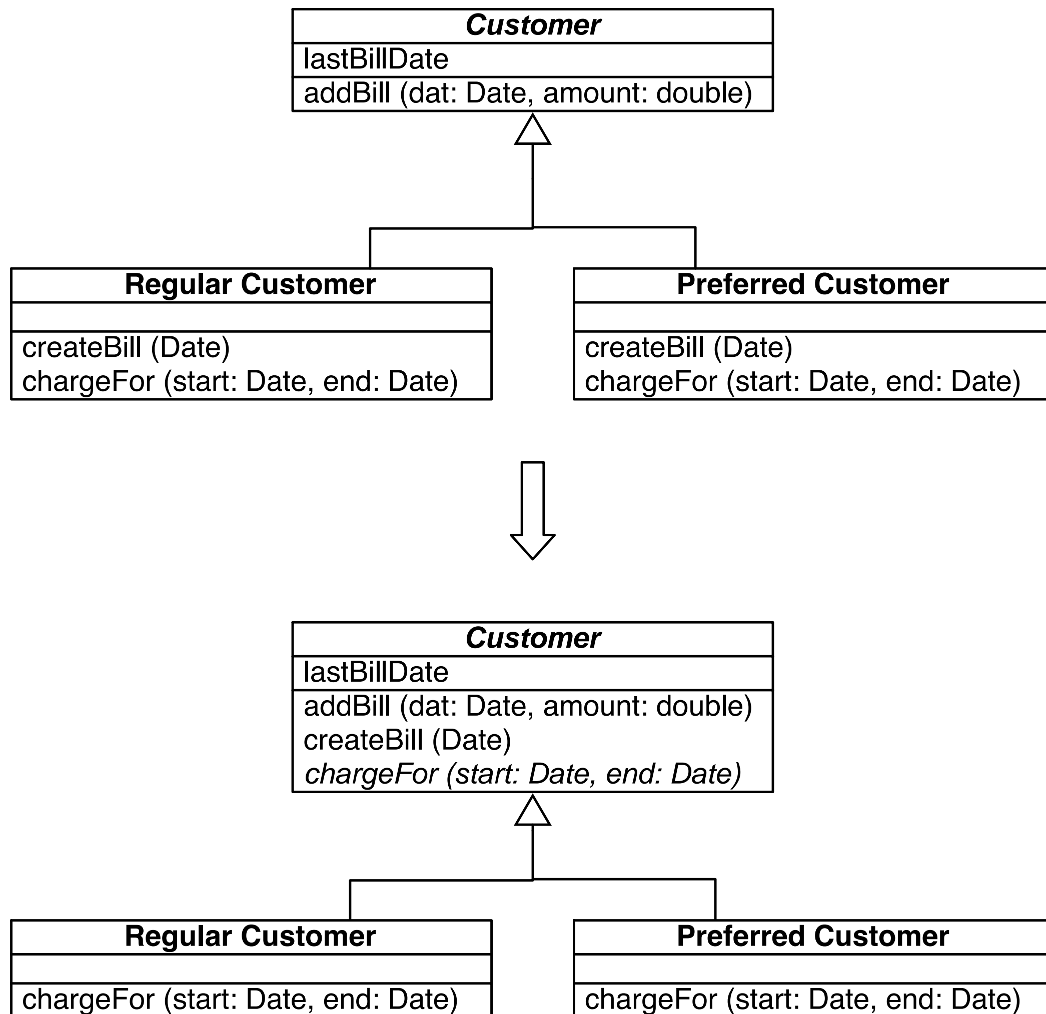

Figure 4.1: *Pull Up Method* refactoring

Example

Consider a customer with two subclasses: regular customer and preferred customer. They differ in the price they are charged on their bills. When a bill is created, the total charge is calculated and noted on the bill. To calculate the charge, a method `chargeFor` is implemented. At the end of the method, a discount percentage is added in the calculation. Each type of customer gets a different discount percentage. Listing 4.1 shows a portion of the code implementing these classes.

```
1  class RegularCustomer extends Customer {
2    // ...
3   double chargeFor(date start, date end) {
4     // ...
5     return result * REGULAR_DISCOUNT;
6   }
7
8    void createBill (Date date) {
9      double chargeAmount = chargeFor(lastBillDate, date);
10     addBill(date, charge);
11   }
12 }
13
14 class PreferredCustomer extends Customer {
15   // ...
16   double chargeFor(date start, date end) {
17     // ...
18     return result * PREFERRED_DISCOUNT;
19   }
20
21   void createBill (Date date) {
22     double chargeAmount = chargeFor(lastBillDate, date);
23     addBill(date, charge);
24   }
25 }
```

Listing 4.1: Code *before* refactoring

We note that both the `RegularCustomer` class and `PreferredCustomer` class have the `createBill` method implemented. Furthermore, both implementations are identical. We remove this code duplication by refactoring using the "Pull Up Method" technique. First, we outline the steps that are generally needed to apply the refactoring.

In this case, we cannot simply move this method up in the hierarchy because this method depends on the `chargeFor` method, which is not part of the superclass `Customer`. First we declare this method on the superclass as abstract. Next, we can copy the `createBill` method from one of the subclasses and remove it

from both subclasses.

The final result of the refactoring is the code sample in listing 4.2. Figure 4.1 already showed an UML diagram of the class before and after applying the refactoring.

```
1  class Customer {
2    // ...
3    abstract double chargeFor(date start, date end);
4
5    void createBill (Date date) {
6      double chargeAmount = chargeFor(lastBillDate, date);
7      addBill(date, charge);
8    }
9  }
```

Listing 4.2: Code *after* refactoring

4.1.5 Tool Support

Manually applying these refactorings proves to be a lot of work and thus very time-consuming. Tool support is available to automate the refactoring process. All the steps in performing a refactoring can be reduced to a few instructions to the tool, greatly improving refactoring speed. In addition, because the process is automated, it can do with less testing. Eliminating tests also speeds up the refactoring process.

Some refactoring tools support “undo” operations. It allows developers to browse their options, trying refactorings as they please, but still be able to roll back to a prior version if they change their mind.

The easier it is to refactor code, the more likely it is that developers will actually do the refactorings. Having tool support will only make it easier for developers to refactor.

4.2 Aspect-aware Object-oriented Refactorings

Refactoring means transforming a program’s source code, and, as such, changing the shadow points in the code. Consequently, the run-time join points of the program will be changed as well. If aspects are not aware of the modifications made, and the pointcut specifications are not adapted, the refactoring will no longer be behavior-preserving. We have to make the refactorings “aspect-aware” [16]. We need to modify the standard OO refactorings to be used in conjunction with AOP. In order to correctly refactor program code, not only the base code has to be transformed, but also all necessary pointcut specifications in the aspect code. Aspects are woven onto a set of join points provided by the base program. Since refactorings are transformations of the base program they change the set of join points. In this section we will discuss the problems that occur when applying object-oriented refactorings to aspect-oriented code.

4.2.1 Example Problem

We use the following example to demonstrate the need for aspect-aware refactorings. Imagine a temperature sensor object that receives a temperature update every ten seconds and a humidity update every second. To be able to calculate average temperature and humidity values, all updates are stored in the database. This persistency is identified as a secondary concern and is placed in the `PersistentTemperatureSensor` aspect. After each call to an update method, the weaver is instructed to insert the value into the database. We encode this into an advice for which the pointcut is shown in listing 4.3.

```
1 pointcut setter(TemperatureSensor ts, double value):
2   target(ts) && args(value) &&
3   call(void TemperatureSensor.set*(double));
```

Listing 4.3: Pointcut for persistent temperatures

If we were to apply any object-oriented refactorings, we have to make sure that the join points described in the pointcut are still covered. Applying "rename method" refactoring requires us to rename the method and all its calls. If we changed the method `setTemperature` to `addTemperature`, the pointcut would no longer cover all join points. If we want to apply this refactoring correctly, we have to adapt the pointcut to include the new join points, resulting in the pointcut in listing 4.4.

```
1 pointcut setter(TemperatureSensor ts, double value):
2   target(ts) && args(value) && (
3   call(void TemperatureSensor.setHumidity(double))) ||
4   call(void TemperatureSensor.addTemperature(double))));
```

Listing 4.4: Adapted pointcut for persistent temperatures

When applying a certain refactoring, the possibly affected join points must be determined. We need to identify the pointcut designators, and the aspects in which they occur, which are used to determine those affected join points. We need to determine how the join points are affected by the refactoring. Certain refactorings change the number of join points, while other will have different context information available at the join points, making it harder to find a suitable pointcut to describe the join points.

4.2.2 Making Object-oriented Refactorings Aspect-aware

We define aspect-aware refactorings as refactorings that can be applied to the base program of an aspect-oriented system [16]. In order to main the behavior-preserving property of refactoring, aspect-aware refactorings not only transform the base program, but also all necessary pointcut specifications.

Hanenberg suggests [16] the following set of enabling conditions to ensure behavior preservation by refactorings in aspect-oriented systems.

- The number of join points addressed by a particular pointcut is not changed after refactoring. If a transformation changes the number of available join points, the pointcut also needs to be transformed to maintain the same number of covered join points.
- Join points covered by a particular pointcut remain in an equivalent position in the programs control flow after refactoring.
- The join point information offered by each pointcut does not decrease. Therefore, aspects of which the behavior varies depending on the information in the join points remain valid.

We discuss the implications of making the refactorings aspect-aware on three object-oriented refactorings. The “rename method”, “extract method” and “move method” refactorings were already discussed in section 4.1.3. We will now concentrate on the changes that need to be made.

Rename Method

Renaming a method means that its method signature is changed. We have to consider the pointcut designators that depend on method signatures. To apply the refactoring we start as we would do in the object-oriented case. We change the method name and rename all method calls to it. We also have to change the method calls in the aspect code. Next we change the signature pattern in the pointcuts that address join points only consisting of this method signature. If the method signature occurs in a collection of join points we have to apply the “composite pointcut” aspect-oriented refactoring. We can now copy the affected pointcut designator and replace the signature pattern. Finally, we compose the pointcuts using disjunction (“or”, AspectJ’s `||` operator).

We also have to check if the new pointcut covers too many join points. In such a case we can remove the superfluous method signatures from the pointcut. Using conjunction (“and”, AspectJ’s `&&` operator) and negation (“not”, AspectJ’s `!` operator), we can remove the unwanted join points from the collection.

Extract Method

The “extract method” refactoring creates a new method and inserts a call to this method where the code was removed. By adding a new method, the number of join points is increased.

Since the control flow within the initial method is changed, pointcuts using the control flow information have to be adapted. A `withincode` pointcut needs to be adjusted to include the join points that have been moved to the new method. It may not, however, include the join points that represent the call of the new method. Using a combination of the `withincode`, `execution` and `cflow` pointcut

designators and the `&&`, `||` and negation operations a new behavior-preserving pointcut can be composed.

We also need to make sure that the new method signature is not included in other pattern-based pointcuts. Remove the method signature from the collection of covered join points by using the conjunction and negation operators.

Move Method

The "move method" refactoring is applied when a method better belongs in a class different from where it is currently defined. The pointcut designators of interest are those which contain a type pattern. Unless the pointcut captures the context at the join point, the following procedure can be applied. Copy the pointcut and replace the type pattern with the new type. Compose the new pointcut with the old pointcut using the disjunction operator.

If the original pointcut makes use of the context at the join point, using the `target` or `this` designators, the following steps have to be taken. Create a new pointcut, based on the affected pointcut and replace the type pattern by the new type. Also change the object type of the pointcuts arguments used by the `target` and `this` designators. Next, copy the pieces of advice applied by the initial pointcut and replace the corresponding pointcut. And last, check whether the pieces of advice are still type-correct. If they are not, introducing or moving further elements can fix the issue.

4.3 Aspect-Oriented Refactorings

In the previous section we made existing object-oriented refactorings aspect-aware by making sure the transformation preserve the behavior provided by the aspect code. This section discusses refactorings that restructure the object-oriented base program using aspect-oriented features, or transform pure aspect code.

Refactoring aspect-oriented systems is possible in three ways [16]. First, refactoring can be used to restructure the base program. Second, refactoring can help to restructure OO code in an aspect-oriented way. And third, refactoring can be applied to the aspect-oriented constructs to improve their comprehensibility and modularity.

4.3.1 Refactorings Catalogue

Monteiro [24] distinguishes three types of aspect-oriented refactorings. We summarize these categories here. In the following section we discuss another refactoring in greater detail.

- **Refactorings for feature extraction** These refactorings deal with moving various elements from their original place in the base object-oriented language into aspects. Inter-type declarations make it particularly easy to move

elements to aspects. From the point of view of the client code, there is no difference between a method declared in its own class, or one introduced by an aspect. When a class declares a field or a method related to a concern other than the primary concern, the declaration should be moved to an aspect. The "*extract feature into aspect*" refactoring is to be used when a feature in the base code that should be evolved separately from the primary code base, is scattered across several units of modularity, such as methods and classes. The feature becomes unpluggable by extracting all the related code into an aspect. In section 4.3.2 we discuss the "method to advice" refactoring in greater detail.

- **Restructuring the internals of aspects** Monteiro [24] notes that aspects resulting from feature extractions are generally badly formed, showing much duplication and inadequate internal structure. These refactorings deal with improving the internal structure of an aspect after all elements from a crosscut are moved into it.

- **Dealing with generalizations** These aspects deal with various "pull up" en "push down" refactorings, similar to the same group of object-oriented refactorings from section 4.1.3. The two previous categories of refactorings leave the inter-aspect structure badly formed. These refactorings improve this structure of the aspect code. E.g. the "extract superaspect" refactoring creates a superaspect from two or more aspects containing similar code and functionality and moves the common features to the superaspect. The "pull up advice" refactoring moves advice to a superaspect, given that the advice is the same in several subaspects and acts on a pointcut declared in the superaspect.

4.3.2 Example: *Method to advice* Refactoring

In this section we discuss the "method to advice" or "extract method calls" refactoring. This refactoring should be used when a certain feature is to be moved to an aspect, but it should be run in all places it currently stands.

Motivation

The refactoring is a combination of a standard OO refactoring and an aspect-oriented refactoring. The OO refactoring "extract method" moves the code into a new method and inserts calls to this method wherever needed. The "extract method calls" refactoring augments this by removing these duplicated method calls in the base code and places a call in an advice body. It makes a pointcut that describes all join points where the method originally is called. Before copying the code fragment, the method's body should be carefully analyzed in order to find a suitable pointcut to capture the exact set of intended join points.

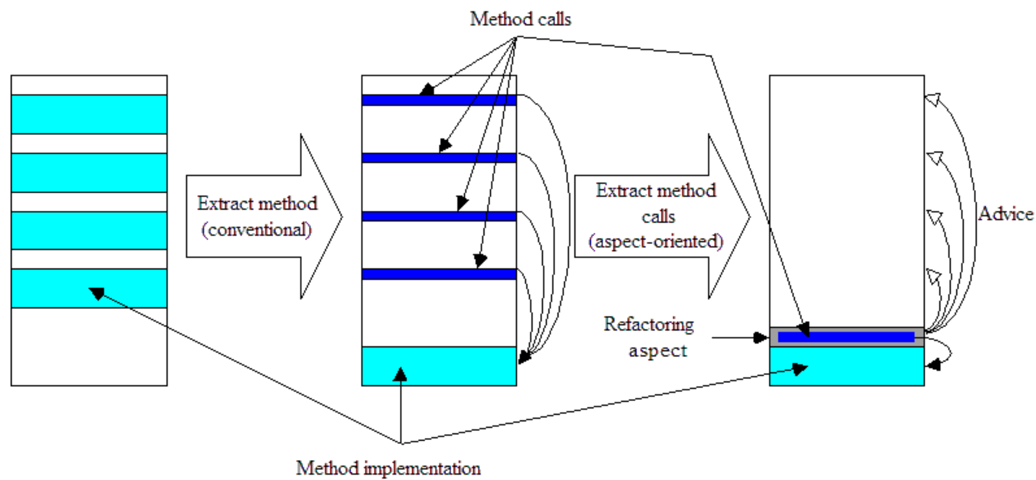

Figure 4.2: Extract method calls refactoring [22]

Figure 4.2 illustrates this process. The left part of the figure shows the base code, before any refactorings are applied. The middle part shows the duplicated method calls after the "extract method" refactoring. Finally, the right part shows the result after the aspect-oriented "extract method calls" refactoring.

Mechanics

The steps needed to complete the "method to advice" refactoring are the following.

1. We introduce a no-op refactoring aspect. The aspect has all the necessary pointcuts, but the advices are left empty. The pointcut simply enumerates the locations of the extracted method calls. IDE support can be used to verify if the pointcut captures the right join points.

2. Next, we introduce the crosscutting functionality. We move the method calls to the advice and pointcuts introduced in the previous step.

3. Make crosscuts pattern-based. While Laddad [22] indicates this last step as optional, Kellens and Gybels [18] stress the importance of simplifying the enumeration-based pointcut. Such pointcuts are too tightly coupled to a specific version of the base program. On each change of the base program, they may need adaptation. Transforming the pointcut into a pattern-based pointcut avoids this problem. It generalizes the pointcut based on the commonalities in the join points. When the program is changed, the pattern-based pointcut automatically captures new join points that match the described pattern.

Example

Consider the example of a bank account object. The `Account` class performs a permission check at the beginning of most methods. The code for this class is shown in listing 4.5. There is not much left for a conventional technique to refactor the permission checks. A method call to `checkPermission` is present in almost every method.

```
1  public class Account {
2    private int _accountNumber;
3    private float _balance;
4
5    public Account(int accountNumber) {
6        _accountNumber = accountNumber;
7    }
8
9    public int getAccountNumber() {
10       AccessController.checkPermission(new
11         BankingPermission("accountOperation"));
12       return _accountNumber;
13   }
14
15   public void credit(float amount) {
16       AccessController.checkPermission(new
17         BankingPermission("accountOperation"));
18       _balance = _balance + amount;
19   }
20
21   public void debit(float amount) throws
22     InsufficientBalanceException {
23       AccessController.checkPermission(new
24         BankingPermission("accountOperation"));
25       if (_balance < amount) {
26         throw new
27           InsufficientBalanceException("Insufficient␣total␣balance");
28       } else {
29         _balance = _balance - amount;
30       }
31   }
32
33    public float getBalance() {
34     AccessController.checkPermission(new
35       BankingPermission("accountOperation"));
36     return _balance;
37   }
```

```
38
39    public String toString() {
40      return "Account:␣" + _accountNumber;
41    }
42  }
```

Listing 4.5: Account class *before* refactoring

To begin the refactoring, we first define a pointcut which captures the join points that need the refactored functionality. In this first step we simply enumerate each of the required methods. The resulting pointcut is shown in listing 4.6.

```
1  permissionCheckedExecution(?jp) if
2    reception(?jp, Account.getAccountNumber).
3  permissionCheckedExecution(?jp) if
4    reception(?jp, Account.credit).
5  permissionCheckedExecution(?jp) if
6    reception(?jp, Account.debit).
7  permissionCheckedExecution(?jp) if
8    reception(?jp, Account.getBalance).
```

Listing 4.6: Enumeration based pointcut

In the following step, we move the code from the base class into the aspect. We also remove the method calls from each of the advised methods. Listing 4.7 shows the aspect with the pointcut and advice. Listing 4.8 shows the `Account` class after refactoring. There is no trace left of the permission check, it is completely moved to the aspect.

```
1  check := Aspect new: 'PermissionCheck'.
2  check before: 'permissionCheckedExecution(?jp),
3                 reception(?jp, ?method)'
4        put:    'AccessController.checkPermission(...)'.
```

Listing 4.7: Crosscut functionality

```
1  public class Account {
2    private int _accountNumber;
3    private float _balance;
4
5    public Account(int accountNumber) {
6      _accountNumber = accountNumber;
7    }
8
9    public int getAccountNumber() {
10     return _accountNumber;
11   }
12
```

```
13    public void credit(float amount) {
14      _balance = _balance + amount;
15    }
16
17    public void debit(float amount) throws
18        InsufficientBalanceException {
19      if (_balance < amount) {
20        throw new
21          InsufficientBalanceException("Insufficient␣total␣balance");
22      } else {
23        _balance = _balance - amount;
24      }
25    }
26
27    public float getBalance() {
28      return _balance;
29    }
30
31    public String toString() {
32      return "Account:␣" + _accountNumber;
33    }
34  }
```

Listing 4.8: Account class *after* refactoring

Fnally, in the last step, we simplify the pointcut. Listing 4.9 shows the simplified aspect.

```
1  check := Aspect new: 'PermissionCheck'.
2  check before: 'methodInClass(?method, Account),
3                 not(equals(?method, toString)'
4                 reception(?jp, ?method)'
5        put:    'AccessController.checkPermission(...)'.
```

Listing 4.9: Simplified pointcut

4.4 Problems with Automating Aspect-Oriented Refactorings

In this section we discuss some problems that occur with aspect oriented refactoring. When refactoring, a call which is not necessarily implemented at the beginning or the end of a method, complicates writing a pointcut which is both behavior-preserving and covers the intent of the original calls.

4.4.1 Flattened Expression Problem

Moving a message call to an advice body in a refactoring should make sure that the call happens at the correct join point. When formulating a pointcut, we have a conflict of interests: we could choose to describe the correct join points or choose to describe the intention of the developer. Following the developers intention does not necessary lead to a pointcut that captures the correct join points. In statements containing nested expressions more than one join point occurs. If we were to refactor the statement proceeding such a nested statement, which join point should we choose? The main expression of the statement is probably the developers original intent, but it is not what actually happens.

```
1 checkPositive(amount);
2 Logger.log("Client updates balance");
3 updateBalance(interestsOn(amount));
```

Listing 4.10: Flattened expression problem

Consider the code fragment in listing 4.10. In this fragment, the developer wants to log before the balance update, but if we were to flatten these statements, the following steps appear.

1. Write to log
2. Calculate interests
3. Update balance

A crosscut that logs before updating the balance, is no longer behavior-preserving. To solve this problem Kellens and Gybels [18] suggest adding static context information to the join points, making it possible to crosscut on that information.

4.4.2 Join Point Choice Problem

When a join point has to be chosen, neither in the beginning nor at the end of a method, there is the choice of picking an after advice on the previous statement, or picking a before advice on the next statement. The human reader can probably easily choose the right pointcut. Automated tools will have to fall back on pattern-based pointcuts to better capture the intention [18].

Tourwé et al. [34] state that the problems with aspect-oriented refactoring are due to the limitations of today's pointcut languages. These languages are not expressive enough to express very complex pointcuts. The pointcuts are very tightly coupled to the applications structure and the developers are forced to deal with the pointcuts at a too low level.

Automating aspect-oriented refactoring is a difficult problem. We cannot simply move code fragments around as most OO refactorings appear to do. The real

problem lies in finding a good pointcut; one that captures the intention of the developer. The pointcut should be as close to the original join points as possible to maintain the behavior-preserving property of refactoring.

4.5 Summary

This chapter discussed the refactoring of programs. We gave an overview of the bad smells that can appear in ill-designed programs and how certain refactorings fix these problems. We discussed how to make regular object-oriented refactorings "aspect-aware" and gave an overview of aspect-oriented refactorings. We noticed that the hardest challenge in aspect-oriented refactoring was to find a good pointcut, that captures the intention of the developer, while preserving program behavior. In the following chapters we discuss a machine learning technique, ILP, and show how it can be used to learn such a pointcut.

Chapter 5

Inductive Logic Programming

In a previous chapter we discussed logic programming. In this chapter we show a machine learning technique to learn logic rules from a collection of facts. We discuss two examples of ILP algorithms: FOIL and Relative Least General Generalization.

5.1 Definitions

Inductive Logic Programming (ILP) deals with systems and general methods that are given examples and produce programs. A system is given a number of positive and negative examples and the logic program that is learned, should distinguish the positive and negative examples as expected. Typically, the obtained programs will then act on new examples, not given during the learning phase.

5.1.1 Definition and Terms

Before we continue with a more formal definition of the ILP problem, we define *completeness* and *consistency* of logic programs.

Consider E^+ a set of positive examples and E^- a set of negative examples. A program P is complete if it covers the complete set of positive examples. A program P is consistent if it does not cover any negative examples.

Definition 5.1. A logic program P is complete, with respect to E^+, if and only if, for all examples $e \in E^+, P \vdash e$.

Definition 5.2. A logic program P is consistent, with respect to E^-, if and only if, for no example $e \in E^-, P \vdash e$.

Bergadano and Gunetti [5] now define the ILP problem as follows.

Definition 5.3. Given a set $\mathcal{P}$ of possible programs, a set E^+ of positive examples, a set E^- of negative examples and a consistent logic program B, such that $B \nvdash e^+$, for at least one $e^+ \in E^+$. Find a logic program $P \in \mathcal{P}$, such that the program $B \cup P$ is complete and consistent.

The input program B is the background knowledge we have about the problem. It describes our knowledge in a number of facts and rules. We impose the condition that at least one of the positive examples cannot follow from the background information. If all positive examples were already covered by the background knowledge, there would be no need to search for the target program.

The set $\mathcal{P}$ is called the hypothesis space. It is an infinite set, containing all possible, correct Horn clause programs. The program P we want to find, is an element of the hypothesis space $\mathcal{P}$, and extends the background program B. It should cover all positive examples in E^+, while not covering any of the negative examples in E^-.

5.1.2 Example

In the following example, we express our knowledge about a family's relationships and want to induce a program that expresses the granddaughter relationship [17, 23]. We will use this example throughout this chapter.

$\mathcal{P}$ = the collection of all correct Horn clauses
E^+ = { `grandDaughter(sharon, victor),`
`grandDaughter(julie, victor)` }
E^- = { `grandDaughter(ellen, victor)` }
B = { `father(bob, tom), father(victor, ellen),`
`father(bob, sharon), female(sharon),`
`mother(ellen, julie), mother(ellen, sharon),`
`female(ellen), female(julie), ...` }
P = { `grandDaughter(?x, ?y)` ← `female(?x),`
`father(?z, ?x), father(?y, ?z),`
`grandDaughter(?x, ?y)` ← `female(?x),`
`mother(?z, ?x), father(?y, ?z),`
`...` }

The set of positive examples E^+ consists of two examples for which the relationship holds. These are two correct instances of the granddaughter relationship. Likewise, there is one example of an incorrect in the set of negative examples E^-. For the background information B we haved expressed facts about the family relationships and gender of the family members. We now wish to find a program P in $\mathcal{P}$ to extend the background knowledge B. The program P is obtained by induction and contains the rules that express the granddaughter relationship. All positive examples have been covered by the program, while it doesn't cover any of the negative examples.

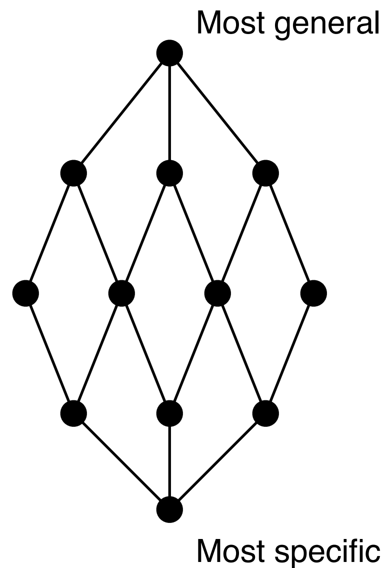

Figure 5.1: Visual representation of the hypotheses space

5.2 Properties of Inductive Algorithms

5.2.1 Top-down vs. Bottom-up

Finding a program in the hypothesis space, means performing a search. We order the hypothesis space by how specific the hypotheses are. At the top is the most general hypothesis, it simply states that everything is true. At the bottom of the hypothesis space is the most specific hypothesis, stating everything is false. In between these two are all other hypotheses. Figure 5.1 depicts the hypotheses space as a lattice.

- **Bottom-up** Bottom-up algorithms start with the most specific hypothesis. At each step in the algorithm, they hypothesis is made more general until all positive examples are covered.
- **Top-down** Top-down algorithms search in the opposite direction. Starting with the most general hypothesis, the algorithm specializes until no more negative examples are covered.

5.2.2 Representation of Background Knowledge

Background knowledge can be available in different ways. We discuss the difference between extensional and intentension knowledge.

- **Extensional** Extensional background knowledge is represented by a collection of facts. The facts define a model of the background we want to use in

the algorithm. An example of extensional background knowledge is shown in the following set. It is based on the example in section 5.1.2, but we have added a `parent` clause. The background information would then be $B = \{$ `father(bob, sharon), mother(ellen, julie), mother(ellen, sharon), parent(bob, sharon), parent(ellen, julie), parent(ellen, sharon), ...` $\}$.

- **Intensional** background knowledge uses a description of the knowledge, rather than enumerating all facts. The background information is represented by a collection of Horn clauses. Not all background information can be provided in this way, so some data is still provided by simple facts. An intensional representation of the previous set is then $B = \{$ `parent(?x, ?y)` $\leftarrow$ `father(?x, ?y), parent(?x, ?y)` $\leftarrow$ `mother(?x, ?y), father(bob, sharon), mother(ellen, julie), mother(ellen, sharon), ...` $\}$.

The extensional background can be generated from the intensional. Taking the facts from the background as input, all resulting facts of the rules in the background information form the extensional background.

5.3 Relative Least General Generalization

In this section and the next we discuss two ILP algorithms in greater detail. The first approach is relative least general generalization, a bottom-up algorithm. The second one is FOIL, which is a top-down algorithm.

The first technique we discuss is Relative Least General Generalization (RLGG). This algorithm uses the fact that induction is no less than the opposite operation of deduction. It uses inverse deduction operators to induce new clauses. RLGG induces new clauses by taking two examples and trying to generalize them in a new clause which is strictly more general and covers both examples.

5.3.1 Definitions

To be sure that the induction process finds rules that are useful and not overly general, a restriction is placed on the induction. We want to find the most specific clause that is more general than the two examples (minimally more general). The following definitions allow us the compare clauses in terms of generality.

Definition 5.4. Clause C **θ-subsumes** (or is a generalization of) a clause D, if there is a substitution θ such that $C\theta \subseteq D$.

For example, clause A θ-subsumes clause B in the following case: $A =$ `mother(?x, ?y)` $\leftarrow$ `father(?x, ?z), spouse(?z, ?y)` and

$B =$ `mother(?x, louise)` $\leftarrow$ `father(?x, bob), spouse(bob, ?y), female(?x)`. If we choose $\theta = \{$`?y/louise`, `?z/bob`$\}$, then $A\theta \subseteq B$.

We refer again to the hypothesis space as depicted in figure 5.1 on page 40. We can see that the lattice form implies that for every pair of clauses, a unique clause exists which is minimal more general than the two clauses. We call this unique clause the least general generalization of the two clauses.

Definition 5.5. C is the **least general generalization** (lgg) of D if C θ-subsumes D and, for every other clause E, if E θ-subsumes D, then it is also the case that E θ-subsumes D.

Before we can compute the least general generalization of two terms, we need to define anti-unification. This operation is the inverse of unification. It operates by comparing the terms occurring at the same position in the two clauses, and replacing them by a new variable if they are different.

Definition 5.6. The **anti-unification** of two terms is the process of obtaining a new term, sharing the commonalities of the two terms, and generalizes the differences.

For example, if we have $A =$ `parent(bob, ellen)` and $B =$ `parent(bob, sharon)`, using the substitutions $\theta_1 = \{$`?x/ellen`$\}$ and $\theta_2 = \{$`?x/sharon`$\}$, we obtain the anti-unification $C =$ `parent(bob, ?x)`.

Using these definitions, we can compute the lgg of two clauses. Consider to two clauses $C_1 \leftarrow A_1, A_2, \ldots, A_n$ and $C_1 \leftarrow B_1, B_2, \ldots, B_m$. We construct the lgg C of the two clauses using the following procedure:

1. The head of the clause C is the anti-unification of the heads of the original clauses C_1 and C_2.

2. The body of the clause C is constructed by the anti-unification of all the terms of the first clause, with all the terms of the second clause: A_i anti-unify with B_j ($\forall i, j : 0 < i < n, 0 < j < m$).

5.3.2 Relative Least General Generalization

Now that we can generalize a set of clauses, we want the incorporate background information into the generalization process. The relative least general generalization (rlgg) of two positive examples is the lgg of the examples with respect to a background model B. Note that the induction of clauses is a specific-to-general search of hypothesis space (bottom-up approach). More formally, rlgg is expressed as follows ($B_\wedge$ denotes the conjunction of all the ground facts in B).

$$rlgg(e_1, e_2, B) = lgg(e_1 \leftarrow B_\wedge, e_2 \leftarrow B_\wedge).$$

5.3.3 Example

In this example we will induce the append relation. Suppose we have to following set of (positive) examples:
{append(<1,2>, <3,4>, <1,2,3,4>),append(<a>, <>, <a>), append(<>, <>, <>), append(<2>, <3,4>, <2,3,4>) }. The rlgg of the first two examples in this set is the θ-lgg of the following two clauses:

```
append(<1,2>,<3,4>,<1,2,3,4>) ←
append(<1,2>, <3, 4>, <1,2,3,4>), append(<a>, <>, <a>),
append(<>, <>, <>), append(<2>, <3, 4>, <2,3,4>).
```

and

```
append(<a>,<>, <a>) ←
append(<1,2>, <3,4>, <1,2,3,4>),append(<a>, <>, <a>),
append(<>, <>, <>), append(<2>, <3,4>, <2,3,4>).
```

The body of the resulting clause consists of 16 literals, constructed by the pair-wise anti-unification of facts in the background knowledge.

```
   append(<?a|?b>,?c,<?a|?d>) ←
append(<1,2>,<3,4>,<1,2,3,4>), append(<?a|?b>,?c,<?a|?d>),
append(?w,?c,?x), append(<?s|?b>,<3,4>,<?s,?t,?u|?v>),
append(<?r|?g>,?k,<?r|?l>), append(<?a>,<>,<?a>),
append(?q,<>,?q), append(<?p>,?k,<?p|?k>), append(?n,?k,?o),
append(?m,<>,?m), append(<>,<>,<>), append(?g,?k,?l),
append(<?f|?g>,<3,4>,<?f,?h,?i|?j>), append(<?e>,?c,<?e|?c>),
append(?b,?c,?d), append(<2>,<3,4>,<2,3,4>)
```

Clearly, this clause contains many redundant literals. Muggleton and Feng [26] calculate that in a background model M, the rlgg of n examples can be of length $|M|^n$. Some techniques exist to reduce the number of unwanted literals in the resulting clauses. First, obviously, removing the ground facts from the clause does not change the logical meaning of the clause, since they are already known to be true. In the next section, we will briefly discuss some other methods to reduce the number of redundant literals.

5.3.4 Reducing the Size of Clauses

Because the size of clauses generated by the rlgg algorithm is so large, we want to reduce the number of redundant literals in these clauses. In this section we briefly discuss three methods for limiting the number of unwanted literals.

- **ij-determination** This technique [26] limits the hypothesis space by limiting the variables that can appear in a clause. The idea is to calculate how dependent variables are on each other ("degree", the *j* component) and how far these dependencies reach ("depth", the *i* component). Limiting the values for i and j decreases the number of literals that can appear in the body of the induces clause.

- **Negative-based reduction** This technique gradually removes literals from a clause, making sure the reduced clause does not cover any literals. The process stops when further reduction doesn't make the clause smaller (removing literals makes the clause cover negatives).

- **Functional reduction** The last technique [29] we discuss puts an additional constraint on Horn clauses. It labels variables of literals as either input or output variables. It creates a functional and/or graph from the unreduced rlgg. By searching the graph and using the input/output information of the variables, a set of variables can be obtained to reduce the clause.

5.4 FOIL

The next algorithm we discuss is FOIL [28]. The hypotheses learned by FOIL are sets of first-order rules, where each rule is similar to a Horn clause. However, there are two differences with general Horn clauses. First, because no literals containing function symbols are allowed, the rules learned by FOIL are more restricted than general Horn clauses (this reduces the complexity of the hypothesis space search). Second, FOIL rules are more expressive because literals appearing in the body of the rules may be negated.

FOIL starts with a general rule and adds literals to it until no more negative examples are covered. At each step it will generate a large collection of literals which are considered for addition. The final choice of which literal is used, is decided by means of a heuristic function.

In this section we discuss the FOIL algorithm, how candidate literals are created and the heuristic used to select the best candidate literal.

5.4.1 Algorithm Overview

The pseudo-code for the FOIL algorithm is shown in figure 5.2. FOIL expects as input a number of examples and a set of background information ($Predicates$). From this input, it will induce a rule for target predicate.

The algorithm consists of a double loop. The outer loop adds new rules. The effect of each new rule is to generalize the current disjunctive hypothesis. It will increase the number of positive instances that are covered. This is a specific-to-general search through the hypothesis space, starting with the empty disjunction and terminating when the set of rules is general enough to cover all positive examples.

The inner loop constructs the rules. It searches a second hypothesis space, consisting of conjunctions of literals, for a conjunction that forms the precondition for the new rule. It starts with most general rule (empty precondition) and adds literals one at a time to specialize the rule until no more negative examples are covered. This part of the search is a general-to-specific search.

FOIL($Target_predicate$, $Predicates$, $Examples$)

1: $Pos \leftarrow Examples$ for which $Target_predicate$ is true
2: $Neg \leftarrow Examples$ for which $Target_predicate$ is false
3: $Learned_rules \leftarrow \{\}$
4: **while** Pos is not empty **do** {learn a new rule}
5: $NewRule \leftarrow$ a new rule for $Target_predicate$ with no preconditions
6: $NewRuleNeg \leftarrow Neg$
7: **while** $NewRuleNeg$ is not empty **do** {specialize $NewRule$}
8: $Candidate_literals \leftarrow$ generate candidate new literals for $NewRule$
9: calculate $Foil_Gain$ for each literal in $Candidate_literals$
10: add literal with highest $Foil_Gain$ to preconditions of $NewRule$
11: $NewRuleNeg \leftarrow$ subset of $NewRuleNeg$ that satisfies $NewRule$ preconditions
12: **end while**
13: $Learned_Rules \leftarrow Learned_Rules \cup \{NewRule\}$
14: $Pos \leftarrow Pos \backslash \{$ members of Pos covered by $NewRule\}$
15: **end while**
16: **return** $Learned_rules$

Figure 5.2: FOIL Algorithm

The combination of these loops result in a set of rules that cover all positive examples (disjunction of the separate rules), while avoiding the negative examples (conjunction of the literals in the rules). In the following sections we discuss how new literals are created and how the algorithm decides which literal is preferred in the search.

5.4.2 Creating Candidate Literals

FOIL chooses new candidate literals based on the literals and variables that are already present in the rule, and on the predicates found in the background information ($Predicates$). Suppose the current rule is $P(?x_1, ?x_2, \ldots, ?x_k) \leftarrow L_1 \ldots L_n$. FOIL now considers the following literals for addition as L_{n+1}.

- $Q(?v_1, \ldots, ?v_r)$, where Q is predicate occurring in $Predicates$ and where all $?v_i (\forall i, 0 < i < r)$ are either new variables or variables already present in the rule. At least one of the variables $?v_i$ has to be present in rule: $\exists j$ with $0 < j < r$ and $?v_j$ is a variable occurring in the rule.
- $Equal(?x_j, ?x_k)$, where $?x_j$ and $?x_k$ are variables already present in the rule.
- The negation of the literals formed in the rules above.

The following example demonstrates the generation of literals for the granddaughter relationship [23].

We want the FOIL algorithm to learn rules for the $GrandDaughter(?x, ?y)$ relationship. The background information consists of facts using the predicates $Father$ and $Female$. FOIL begins the search process with a newly created rule, the most general rule: $GrandDaughter(?x, ?y) \leftarrow$. This means that the relationship is true for any $?x$ and $?y$. FOIL specializes the initial rule and generates the following literals as candidates: $Female(?x)$, $Female(?y)$, $Father(?x, ?y)$, $Father(?y, ?x)$, $Father(?x, ?z)$, $Father(?z, ?x)$, $Father(?y, ?z)$, $Father(?z, ?y)$, $Equal(?x, ?y)$, and all negations of these literals, bringing the total to 16 candidate literals. Suppose FOIL chooses $Father(?y, ?z)$ as the next literal. The rule is now $GrandDaughter(?x, ?y) \leftarrow Father(?y, ?z)$. In the next step, FOIL considers all candidates from the previous step, plus the following additional literals $Female(?z)$, $Equal(?z, ?x)$, $Equal(?z, ?y)$, $Father(?z, ?w)$, $Father(?w, ?z)$, and their negations. These literals and the new variable $?w$ are allowed because the variable $?z$ was introduced in the previous step.

The algorithm continues adding new literals until no more negative examples are covered by the rule. The examples that are correctly classified by the generated rule are removed from the set of positive examples. FOIL continues adding new rules until all positive examples are covered.

5.4.3 Selecting the Best Candidate

At each step, FOIL has a large number of candidate literals to choose from. It uses a performance measure $Foil_Gain$ to select the best literal. The value for $Foil_Gain$ is calculated for each candidate literal, and the literal with the highest value is considered the best candidate. $Foil_Gain$ is defined by the following equation.

$$Foil_Gain(L, R) = t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)$$

$Foil_Gain$ calculates the performance of the rule R, extended with the literal L. FOIL generates all possible variable bindings to the rule. It counts the number of bindings that exist in the background data (positive bindings, denoted by p_1) and the number of bindings that do not exist in the background (negative bindings, n_1). These number are compared to the number of positive (p_0) and negative bindings (n_0) of the rule, before it was extended with the literal L. Finally, the number of bindings that remain positive (t) after adding the literal to the rule is factored in.

The following example shows how the $Foil_Gain$ calculation is performed on the $GrandDaughter$ relationship from the previous examples.

Assume the following input: $GrandDaughter(victor, sharon)$, $Father(sharon, bob)$, $Father(tom, bob)$, $Father(bob, victor)$, $Female(sharon)$. To calculate the values for the variables is the $Foil_Gain$ formula, we use the constants in the background information. We generate bindings of the form $\{?x/bob, ?y/sharon\}$,

	p_0	n_0	p_1	n_1	t	$Foil_Gain$
$Equal(?x, ?y)$	1	15	0	4	0	$-\infty$
$Female(?x)$	1	15	0	4	0	$-\infty$
$Father(?x, ?z)$	1	15	0	12	0	-2.33
$Father(?z, ?x)$	1	15	0	8	0	$-\infty$
$Father(?y, ?z)$	1	15	1	11	1	0.415

Table 5.1: $Foil_Gain$ values for candidate literals in the first step

$\{?x/bob, ?y/tom\}, \ldots$. Of the 16 possible bindings, only 1 binding is classified as positive: $\{?x/victor, ?y/sharon\}$. This means that the values for p_0 and n_0 in the first calculations of $Foil_Gain$ are $p_0 = 1$ and $n_0 = 15$. Table 5.1 lists more values for some candidate literals. From the table it is clear that $father(?y, ?z)$ has the highest value for $Foil_Gain$, and will be chosen as the first literal.

5.5 Summary

This chapter gave an overview of inductive logic programming and discussed two ILP algorithms: FOIL and Relative Least General Generalization. In the next chapter we will use ILP to induce a pattern-based pointcut from the information available in a Java program.

Chapter 6

Inducing pattern-based pointcuts

In chapter 4 we concluded that the biggest problem in automating aspect-oriented refactoring was to transform an enumeration-based pointcut into a correct pattern-based, intensional pointcut. In this chapter we show a technique to induce such a pointcut.

6.1 Overall Approach

In chapter 3 we defined a pointcut language which allowed us to formulate expressions as logic queries over join points occurring in a program. In chapter 4 we concluded that the biggest problem in automating aspect-oriented refactoring was to find a correct pattern-based, intensional pointcut. In the previous chapter we discussed inductive logic programming, a reasoning technique which aims to derive a logic query for a desired set of solutions. Given a collection of examples, ILP will search for an expression that covers all the positive examples, but which avoids over-generalization by taking the negative examples in account.

We map the problem of expressing pointcuts by means of join points, to the problem of inducing a logic rule from a set of facts. We will be able to induce a pointcut from a set of join points and background information about the program.

To correctly induce logic programs we need a set of examples, the training data, and a background theory, containing information that can be used in the induction process. This section shows how to map the information available from logic meta-programming to what is needed for ILP.

The desired solution for the ILP algorithm is a pointcut, a rule which covers all join points on which we want to weave. As input to the algorithm, we give it the collection of join points that we want to cover. The background theory contains information about the program and about the shadow points. This means that it contains information about class hierarchies, what classes implement which methods, where methods are called, *etc*. The negative examples are implicitly defined as all other join points in the system.

The basic principle of applying ILP is to generalize the shadow points into

pointcuts that cover all wanted join points, but do not cover any unwanted join points. Generalization boils down to introducing variables to replace ground facts. The following example demonstrates this. Suppose we want to cover all `toString` methods in the following background theory: `methodInClass(toString, ClassA)` and `methodInClass(toString, ClassB)`. Introducing a variable generalizes this to `methodInClass(toString, ?class)`, covering both facts from the background.

6.2 Induction Algorithm

In chapter 5 we discussed two induction algorithms. Each algorithms has its own advantages and shortcomings. In this section we learn which algorithm suits the specific needs for learning pointcuts best.

6.2.1 Available Information

The logical facts we reason about come from Java source. We scan a complete source file, or a set of files, for background information. The information that is available is listed here. It should be noted that the `?jp` variables do not actually represent join points, which happen at run-time, but "join point groups": the set of join points that can be encountered while executing, it is the static representation of a join point. Aside from the information about the join point groups, the background theory also includes static information about the program.

- `inMethod(?jp, ?method, ?class)` The join point occurs in the method `?method` in the class `?class`.
- `reception(?jp, ?method)` The join point occurs at the reception of a method call `?method`.
- `isSendOf(?jp, ?method)` Generated when a statement or expression is a call to the method `?method`.
- `isAssignment(?jp, ?variable)` Generated when a statement performs an assignment to the variable `?variable`.
- `instanceVariable(?variable, ?class)` `?variable` is declared as an instance variable in the class `?class`.
- `methodReturns(?method, ?type)` The method `?method` returns a value of the type `?type`.
- `methodInClass(?method, ?class)` The method `?method` is defined in the class `?class`.
- `classExtends(?class, ?superclass)` The direct superclass of `?class` is `?superclass`.

- `classImplements(?class, ?interface)` The class `?class` implements the interface `?interface`.
- `classInPackage(?class, ?package)` The class `?class` is contained in the package `?package`.

Usually, only a very small number of join points is identified as the points we want to weave on. The negative examples are implicitly generated from all other join points in the system. We thus have a small set of positive examples, a large background theory and a large number of negative examples.

6.2.2 Algorithms Compared

We take a look at the FOIL and RLGG algorithms and observe how they fulfill our requirements from the previous section.

- **RLGG** This algorithm induces the clauses by generalizing the positive examples. RLGG will never add negative conditions to ensure no negative examples are covered. The algorithm will find an enumeration of ground facts, rather than a general rule with a few negative examples as exceptions. The size of the clauses will be very large.
- **FOIL** The FOIL algorithm adds rules until all positive examples are covered. Constructing these rules is guided by negative examples. It will keep adding the most promising literals, based on the $Foil_Gain$ performance measure, to the rule, until no more negative examples are covered. A clever choice of negative examples means we obtain a good result, early in the process.

Gybels and Kellens [15] made use of the RLGG algorithm to uncover pointcuts in a Smalltalk image. They note that the algorithm lacks for this kind of learning, for the reasons listed above. In our implementation we use the FOIL algorithm. The following section will show how it learns a simple pointcut from a Java source file.

6.3 Example: `stateChanges` Rule

In this section we demonstrate how a simple rule is induced. We start from a simple Java class and will incrementally change it to show how the resulting rules change as well.

6.3.1 Basic Class and Positives

```
1 class Counter {
2   int count;
3
```

```
4   int increment() { count = count + 1; return count; }
5   int decrement() { count = count - 1; return count; }
6
7   int value() {
8     int result;
9     result = count;
10    return result;
11  }
12 }
```

Listing 6.1: Base `Counter` class

We list the background information that is obtained from this class.

1. reception(jp1,increment)
2. inMethod(jp2,increment,Counter))
3. isAssignment(jp2,count)
4. inMethod(jp3,increment,Counter)
5. inMethod(jp4,increment,Counter)
6. inMethod(jp5,increment,Counter)
7. inMethod(jp6,increment,Counter)
8. inMethod(jp7,increment,Counter)
9. reception(jp8,decrement)
10. inMethod(jp9,decrement,Counter)
11. isAssignment(jp9,count)
12. inMethod(jp10,decrement,Counter)
13. inMethod(jp11,decrement,Counter)
14. inMethod(jp12,decrement,Counter)
15. inMethod(jp13,decrement,Counter)
16. inMethod(jp14,decrement,Counter)
15. reception(jp15,value)
16. inMethod(jp16,value,Counter)
17. inMethod(jp17,value,Counter)
18. isAssignment(jp17,result)
19. inMethod(jp18,value,Counter)
20. inMethod(jp19,value,Counter)
21. inMethod(jp20,value,Counter)
22. instanceVariable(count,Counter)
23. methodReturns(increment,int)
24. methodReturns(decrement,int)
25. methodReturns(value,int)
26. methodInClass(increment,Counter)
27. methodInClass(decrement,Counter)
28. methodInClass(value,Counter)
29. classExtends(Counter,Object)
30. classInPackage(Counter,Root)

6.3.2 Simple `stateChanges` Rule

We want to find a rule that covers all methods that change the state of an object. We identify the methods `increment` and `decrement` as positive examples. The resulting rule is shown in listing 6.2. A method is state-changing if one of the join points inside it perform an assignment to the `count` variable.

```
stateChanges(increment)
stateChanges(decrement)
```

```
1 stateChanges(?V1) if
2   inMethod(?V2,?V1,?V3),
3   isAssignment(?V2,count)
```

Listing 6.2: Basic `stateChanges` rule

We now try to expand this rule by adding more methods to the `Counter` class. We first add a second instance variable to the class. This second instance variable has its own methods for updating. Listing 6.3 shows the added instance variable and methods. We also add the new methods to the positive examples.

```
stateChanges(increment1)
stateChanges(decrement1)
```

```
1   int count1;
2   int increment1() { count1 = count1 + 1; return count1; }
3   int decrement1() { count1 = count1 - 1; return count1; }
4   int value1() {
5     int result;
6     result = value() + count1;
7     return result;
8   }
```

Listing 6.3: Second counter in `Counter` class

The new rule, shown in listing 6.4, now implies that a method is state-changing if it includes a join point which performs an assignment to an instance variable.

```
1 stateChanges(?V1) if
2   inMethod(?V2,?V1,?V3),
3   isAssignment(?V2,?V4),
4   instanceVariable(?V4)
```

Listing 6.4: `stateChanges` rule if instance variable changed

6.3.3 Recursive `stateChanges` Rule

We now add extra methods for changing the counters by a number of steps at a time. To implement these methods, we call the methods that are already available. Listing 6.5 shows the implementations. Again, we need to add these methods to the positive examples as well.

```
stateChanges(incrementBy)
stateChanges(decrementBy)
stateChanges(increment1By)
stateChanges(decrement1By)
```

```
int incrementBy(int x) {
  for (int i = 0; i < x; i++)
    increment();
  return count;
}

int decrementBy(int x) {
  for (int i = 0; i < x; i++)
    decrement();
  return count;
}

int incrementBy1(int x) {
  for (int i = 0; i < x; i++)
    increment1();
  return count1;
}

int decrementBy1(int x) {
  for (int i = 0; i < x; i++)
    decrement1();
  return count1;
}
```

Listing 6.5: Extra method calls for `Counter` class

The resulting rule now is a recursive rule. As in the previous example, a method is state-changing if it includes an assignment join point. A second rule now implies that a method is also state-changing if one of its join points is a message send of a state-changing method. The rule is shown in listing 6.6.

```
stateChanges(?V1) if
  inMethod(?V2,?V1,?V3),
  isAssignment(?V2,?V4),
  instanceVariable(?V4)
stateChanges(?V1) if
  inMethod(?V2,?V1,?V3),
  isSendOf(?V2,?V4),
  stateChanges(?V4)
```

Listing 6.6: Recursive `stateChanges` rule

6.3.4 Pointcuts

We now induced a rule that identifies state-changing methods. However, this rule is not immediately suited for use in a pointcut language because the variables in

the head of the rules do not unify with join points but with methods. A simple extension of the rule results in a pointcut that can be used:

```
1 stateChangingPoints(?jp) if
2   reception(?jp, ?method), stateChanges(?method).
```

Listing 6.7: `stateChanges` pointcut

6.4 Summary

In this chapter we showed how we can map the problem of finding a pattern-based pointcut to learning a first-order logic rule from a set of examples plus background information. We compared two ILP algorithms and showed that FOIL is the better choice for this particular problem. We gave an example in which we showed how a pattern-based definition for a rule is learned. In the next chapter we will show how the algorithm finds a pointcut from a collection of join points.

Chapter 7

Experiments

In this chapter we demonstrate the induction of pattern-based pointcuts. We include two examples that show the refactoring of the change notification call in the Observer design pattern. We discuss the rules that are generated by our implementation.

7.1 Refactor Observer Methods

In the first experiment we induce a proper pattern-based pointcut from a set of joinpoints. We show the refactoring of the change notification in the "Observer" design pattern [13]. This design pattern implements a kind of change notification between objects. One object in the pattern manages the notifications and decides if some action is needed. The subjects register themselves with this object and call an update method when their state has changed. We assume a basic `Point` class for this experiment which is the subject of such an observer object. The code for the `Point` class is shown in listing 7.1. We assume that all observer-related code is already implemented in the `Observable` interface and that the implementing classes only need to worry about the `update` method. The `Point` class contains calls to this method in its setter methods.

```
1  class Point implements Observable {
2    int x;
3    int getX() { return x; }
4    void setX(int v) {
5      x = v;
6      self.update();
7    }
8
9    int y;
10   int getY() { return y; }
11   void setY(int v) {
12     y = v;
```

```
13      self.update();
14    }
15
16    void move(int dX, int dY) {
17      setX(getX() + dX);
18      setY(getY() + dY);
19    }
20
21    void draw(Graphics g) {
22        g.drawRect(getX(), getY(), 1, 1);
23    }
24
25    void update() {
26      Observer.notifyChanged(this);
27    }
28  }
```

Listing 7.1: Basic `Point` class

7.1.1 Refactoring with basic information

We would like to move the "change notification" concern to an aspect. This means that in this fragment of code, the calls to the `update` method are to be extracted using the "extract method call" refactoring. We show the steps that are to be taken.

1. We introduce a no-op refactoring aspect. We create a pointcut which simply enumerates the locations of the extracted method calls.

2. Introduce the crosscutting functionality. We place the method calls in the advice and pointcuts introduced in the previous step. We remove the calls from the base code.

3. Make pointcuts pattern-based.

Figure 7.1 shows all available background information that can be generated from this program. We enumerate the join points that can occur in the surroundings of the method call again here. After removing the call to the `update` method, we need to create a pattern-based pointcut. We have automated this final step.

1. reception(jp4,setX)
2. isAssignment(jp5,x)
3. inMethod(jp5,setX,Point)
4. reception(jp10,setY)
5. isAssignment(jp11,y)
6. inMethod(jp11,setY,Point)

The `jp4` join point is the reception of the `setX` message. Join point `jp5` is the execution of the assignment statement in the `setX` method. We have the same

- instanceVariable(y,Point)
- instanceVariable(x,Point)
- methodInClass(draw,Point)
- methodReturns(draw,void)
- methodReturns(getX,int)
- methodReturns(setX,void)
- methodReturns(getY,int)
- methodReturns(setY,void)
- methodReturns(move,void)
- methodInClass(getX,Point)
- methodInClass(setX,Point)
- methodInClass(getY,Point)
- methodInClass(setY,Point)
- methodInClass(move,Point)
- methodInClass(update,Point)
- methodReturns(update,void)
- reception(jp1,getX)
- reception(jp4,setX)
- reception(jp7,getY)
- reception(jp10,setY)
- reception(jp13,move)
- reception(jp24,draw)
- reception(jp31,update)
- classInPackage(Point,Root)
- inMethod(jp16,move,Point)
- inMethod(jp17,move,Point)
- inMethod(jp18,move,Point)
- inMethod(jp19,move,Point)
- inMethod(jp20,move,Point)
- inMethod(jp21,move,Point)
- inMethod(jp22,move,Point)
- inMethod(jp23,move,Point)
- inMethod(jp25,draw,Point)
- inMethod(jp26,draw,Point)
- inMethod(jp6,setX,Point)
- inMethod(jp8,getY,Point)
- inMethod(jp9,getY,Point)
- inMethod(jp2,getX,Point)
- inMethod(jp3,getX,Point)
- inMethod(jp5,setX,Point)
- inMethod(jp11,setY,Point)
- inMethod(jp12,setY,Point)
- inMethod(jp14,move,Point)
- inMethod(jp15,move,Point)
- classExtends(Point,Object)
- inMethod(jp27,draw,Point)
- inMethod(jp28,draw,Point)
- classImplements(Point,Observable)
- inMethod(jp29,draw,Point)
- inMethod(jp30,draw,Point)
- isSendOf(jp14,setX)
- isSendOf(jp15,getX)
- isSendOf(jp18,setX)
- isSendOf(jp19,setY)
- isSendOf(jp20,getY)
- isSendOf(jp23,setY)
- isSendOf(jp25,drawRect)
- isSendOf(jp26,getX)
- isSendOf(jp27,getY)
- isSendOf(jp30,drawRect)
- inMethod(jp32,update,Point)
- isSendOf(jp32,notifyChanged)
- inMethod(jp33,update,Point)
- inMethod(jp34,update,Point)
- isSendOf(jp34,notifyChanged)
- isAssignment(jp5,x)
- isAssignment(jp11,y))

Figure 7.1: Background information from basic `Point` class

information about the join points `jp10` and `jp11` of the `setY` method. We have to describe a call at the end of each method, so we can choose two join points to generate the pointcut with. We describe each combination of the four join points.

- **`jp4` and `jp10`** or **`jp5` and `jp11`** The two reception join points or the two assignment join points. We describe the search for a pattern in these combinations in the following sections.
- **`jp4` and `jp11`** or **`jp5` and `jp10`** A reception join point from one method and an assignment join point from the other method. We do not consider these options, because the algorithm will fail to uncover a pattern here.

Reception Join Points

We are inducing the `target` rule. Our algorithm uncovers a pattern in the join points we give as input. We give the reception join points `jp4` and `jp10` as positive examples to the algorithm. The rule that results is shown in listing 7.2.

```
target(jp4)
target(jp10)
```

```
1 target(?V1) if
2   reception(?V1,?V2),
3   reception(?V1,setX)
4 target(?V1) if
5   reception(?V1,setY)
```

Listing 7.2: Rule from reception join points

This rule simple states that a target is a reception join point of either the `setX` or the `setY` method.

Assignment Join Points

We are inducing the `target` rule again. This time we identify the assignment join points `jp5` and `jp11` as positive examples. The resulting rule is shown in listing 7.3.

```
target(jp5)
target(jp11)
```

```
1 target(?V1) if
2   isAssignment(?V1,?V2),
3   instanceVariable(?V2,Point)
```

Listing 7.3: Rule from assignment join points

The `target` rule now implies that a join point is a target, if it performs an assignment to an instance variable of the `Point` class.

7.1.2 Refactoring with Extra Information

We add extra information to the class and see how the resulting rules evolve. We extend the `Point` class with a color attribute. The color of the point is stored in the `color` instance variable and new methods are introduced to set and get its value. Another instance variable `dirty` remembers if the point needs to be drawn the next time the `draw` method is called. It is used to indicate that the point can be removed from the display, but it can wait until the next refresh. We do not want to notify the observer when this variable changes.

The pointcut that we want to induce can no longer say that the assignment to an instance variable triggers the call to the `update` method. And clearly the `dirty` instance variable is the exception to the rule. We show which rules are induced by the algorithm.

Changes to the `Point` Class

We base ourselves on the `Point` class from listing 7.1 on page 55, but we extend the class with these new methods. Note that the definition of the `draw` method is changed as well.

```
1  void draw(Graphics g) {
2    if (not(isDirty()))
3      g.drawRect(getX(), getY(), 1, 1);
4  }
5
6  boolean dirty;
7  boolean getDirty() { return dirty; }
8  boolean setDirty(boolean v) { dirty = v; }
9  void markDirty() { setDirty(true); }
10 boolean isDirty() { return getDirty(); }
11
12 Color color;
13 Color getColor() { return color; }
14 void setColor(Color v) { color = v; self.update(); }
```

Listing 7.4: Additions to `Point` class

We want to remove the call to the `update` method on line 14 as well. Next to the join points we already listed in the previous section, we also consider the join points surrounding the call on line 14. The background information is again extended with a large number of facts about new join points, but we only show the new information about the related join points.

7. reception(jp53,setColor)
8. isAssignment(jp54,color)
9. inMethod(jp54,setColor,Point)

We can now make 8 combinations with these join points, but again we only consider the two where the algorithm will most likely find a pattern.

Reception Join Points

We are inducing the `target` rule again. We input the reception join points `jp4`, `jp10` and `jp53` as positive examples to the algorithm. The rule that results is shown in listing 7.5.

```
target(jp4)
target(jp10)
target(jp53)
```

```
1 target(?V1) if
2   reception(?V1,?V2),
3   reception(?V1,setX)
4 target(?V1) if
5   reception(?V1,?V2),
6   reception(?V1,setY)
7 target(?V1) if
8   reception(?V1,setColor)
```

Listing 7.5: Rule from reception join points

This rule is again an enumeration of the method names. To produce a pattern-based pointcut based on the reception join points requires very long rules. The FOIL algorithm has an inductive bias towards shorter rules. The $FOIL_Gain$ performance measure used to select the candidates implies a best-first search approach, generally resulting in shorter rules.

Assignment Join Points

This time we identify the assignment join points `jp5`, `jp11` `jp54` as positive examples of the `target` rule. The result of the algorithm is shown in listing 7.6.

```
target(jp5)
target(jp11)
target(jp54)
```

```
1 target(?V1) if
2   isAssignment(?V1,?V2),
3   instanceVariable(?V2,Point),
4   not(isAssignment(?V1,dirty))
```

Listing 7.6: Rule from assignment join points

This `target` rule identifies a join point as a target, if it performs an assignment to an instance variable of the `Point` class, except if the assignment is to the `dirty` instance variable.

7.2 Keywords in methodnames

In the next experiment we use extra background information to reason about. We notice that the common Java convention to use "CamelCase" to name methods, can identify namegiving patterns in method names. CamelCase is the name given to the practice of writing compound words, joined without spaces, and each word capitalized within the compound [4].

In this experiment we analyzed the method names and if they where named according to the CamelCase convention, we added this information to the background theory.

- `methodKeyword(?method, ?keyword)`. E.g. when a method named `longName` is encountered, we add these facts to the background: `methodKeyword(longName, long)` and `methodKeyword(longName, name)`.

7.2.1 `Address` Class

We use a class which represents an `Address`. It contains a streetname and number, a city and a zipcode (listing 7.7). After changing any of the instance variables, we notify any observers of a state change. Again, we would like to refactor the `update` method calls on lines 6, 13, 20 and 27.

```
1  class Address implements Observable {
2    String street;
3    void getStreet() { return street; }
4    void setStreet(String v) {
5      street = v;
6      this.update();
7    }
8
9    int number;
10   int getNumber() { return number; }
11   void setNumber(int v) {
12     number = v;
13     this.update();
14   }
15
16   String city;
17   String getCity() { return city; }
18   void setCity(String v) {
19     city = v;
20     this.update();
21   }
22
```

```
23    int zipcode;
24    int getZipcode() { return zipcode; }
25    void setZipcode(int v) {
26      zipcode = v;
27      this.update();
28    }
29
30    String toString() {
31      String result;
32      result = street + number + "\n" + zipcode + city;
33      return result;
34    }
35
36    void update() {
37      Observer.notifyChanged(self);
38    }
39  }
```

Listing 7.7: `Address` class

The background information from the program is comparable to the previous example. We only show the new information about the compound words in the `methodKeyword` facts.

1. methodKeyword(getStreet,get)
2. methodKeyword(getStreet,street)
3. methodKeyword(setStreet,set)
4. methodKeyword(setStreet,street)
5. methodKeyword(getNumber,get)
6. methodKeyword(getNumber,number)
7. methodKeyword(setNumber,set)
8. methodKeyword(setNumber,number)
9. methodKeyword(getCity,get)
10. methodKeyword(getCity,city)
11. methodKeyword(setCity,set)
12. methodKeyword(setCity,city)
13. methodKeyword(getZipcode,get)
14. methodKeyword(getZipcode,zipcode)
15. methodKeyword(setZipcode,set)
16. methodKeyword(setZipcode,zipcode)
17. methodKeyword(toString,to)
18. methodKeyword(toString,string)

7.2.2 Refactoring the `update` Call

In the example we can identify 8 join points where we could weave the method call. We could try to uncover a pattern in the possible 16 combinations of these join points, but again, only two combinations are valid: all reception join points, or all assignment join points. We now discuss the results.

Reception Join Points

We identify the reception join points of the setter methods in the background theory and input them as positive examples for the `target` rule.

```
reception(jp4,setStreet)
reception(jp10,setNumber)
reception(jp16,setCity)
reception(jp22,setZipcode)

target(jp4)
target(jp10)
target(jp16)
target(jp22)
```

The rule for `target` that is induced is shown in listing 7.8. It describes target join points as reception join points, where the message that is received, contains a "set" keyword.

```
1 target(?V1) if
2   reception(?V1,?V2),
3   methodKeyword(?V2,set)
```

Listing 7.8: Rule from reception join points

Assignment Join Points

The second option is to pass the assignment join points as positive examples for the `target` rule.

```
inMethod(jp5,setStreet,Address)
isAssignment(jp5,street)
inMethod(jp11,setNumber,Address)
isAssignment(jp11,number)
inMethod(jp17,setCity,Address)
isAssignment(jp17,city)
inMethod(jp23,setZipcode,Address)
isAssignment(jp23,zipcode)

target(jp5)
target(jp11)
target(jp17)
target(jp23)
```

The resulting rule (listing 7.9) simply describes all join points that perform an assignment to an instance variable of the `Address` class. The information about the keywords was not used in this rule.

```
target(?V1) if
  isAssignment(?V1,?V2),
  instanceVariable(?V2,Address)
```

Listing 7.9: Rule from assignment join points

7.3 Future work

The examples from the previous section are only small examples. Based on the literature on inductive logic programming, we expect that for larger programs some problems could arise. We discuss the problems that can occur and how they could be solved.

- The algorithm always finds a correct pointcut, but it will often use predicates that lead to strange looking rules. Although these rules are correct, they fail to describe the pattern in the positive join points. This problem could arise when multiple candidates have the same gain value: at that time, they are equally good. We can avoid this problem by adding weights to the predicates. These weights are multiplied by the gain value and result in a higher value for candidates using certain predicates. These weights guide the search by adding more value to some preferred predicates.
- Some resulting rules may contain some noise, even after calculating in a weight for certain predicates. Some manual adaption of the rules avoid this problem. A possible solution to this problem could be to analyze the rules and applying heuristics to discover bad smells in the rules. If these are detected, they could be returned to the user. This extra information could help in the adaptation of the rule. Some common bad smells in the rule could be handled by other tool support to enhance the rules.

7.4 Conclusion

In this chapter we validated our claim that pattern-based pointcuts can be generated from the logic representation of a Java program. We implemented the inductive logic programming algorithm FOIL and used it to produce the pointcuts. We have illustrated this with the refactoring of a method call of the commonly used Observer design pattern, moving the "change notification" crosscutting concern from the base code to aspect code.

We noticed that the quality of the background theory is very important to induce good pointcuts. It becomes easier to uncover a pattern if more background information is available. We demonstrated this by gradually extending a class and showing how the rules improve when more information becomes available. More information leaves less room for errors: the induced rules will not be too general.

We noticed that with less background information, the rules would be overly general. Even though the rules were correct at the time of generation, they would cover too many join points if the source was extended.

This thesis's major contribution is showing that the FOIL algorithm can uncover patterns in a given set of join points and can produce pattern-based pointcuts that capture all given join points.

Chapter 8

Conclusions

This chapter concludes the dissertation. In this chapter we summarize the discussion of the previous chapters. We present our conclusion and show the technical contributions made to support our thesis. Finally, we discuss some possible future work on the subject.

8.1 Summary

When programming an application, a large number of requirements are to be met. We decompose the system in smaller parts, and let each part implement a specific subset of the requirements. Because each part addresses only a specific concern, developers can focus on this single concern and temporarily forget about the others. We call this "separation of concerns". An ideal decomposition puts each concern in a separate module, and lets each module handle only a specific concern.

Object-oriented programming languages have some limitations which inhibit good separation of concerns. These languages offer a good decomposition along a single dimension of concern: data types. This leads to certain concerns crosscutting the program, which means they become scattered over the code and entangled with other concerns. Aspect-oriented programming introduces a new modularization mechanism that is able to separate these crosscutting concerns from other concerns, and contains them in their own module. These modules specify for themselves at which point they need to be called. A pointcut expression describes a set of join points, key events in the execution of a program, at which time the aspect weaver should insert the execution of the aspect.

Because of the increased readability, maintainability and other "-ilities" of aspect-oriented software, we want to transform legacy object-oriented programs to use aspects. We use refactorings to gradually transform the source code, while preserving the behavior.

The "method to advice" refactoring transforms the base code by taking a code fragment and moving it to an advice. It defines a pointcut which describes the join points where the code fragment was originally executed. The biggest chal-

lenge of this refactoring is to find a good pointcut definition that makes sure the refactored code will be called where it was removed, and expresses the developer's intention. Existing refactorings limit themselves to enumeration-based pointcuts. These pointcuts are too tightly coupled to a specific revision of the base program. Whenever the program is changed, these pointcuts may need adaptation. Transforming the pointcut into a pattern-based pointcut avoids this problem. It generalizes the pointcut based on the commonalities in the join points. Even when the base program is changed, new join points that match the described pattern will automatically be captured by the pointcut.

Automating the process of finding a pattern-based pointcut is difficult because of some problems that inhibit finding a good description of the needed join points. When the targeted join points occur at neither the beginning nor the end of a method, the description will have to be based on its surroundings. Automated tools cannot interpret the names that are given to methods and variable names in the code, and are thus less informed than humans. It is much harder for automated tools to find a good pointcut description. Looking for a pattern in these names can lead to good intensional, pattern-based pointcuts. The "flattened expression" problem appear when a pointcut has to be found in the proximity of a nested expression. Nested expressions provide multiple join points, and while the developer can mean to put a particular feature right before the top statement, automated tools that study the join point graph will not see this. They will use the first join point that occurs, which always is the most deeply nested expression.

In this dissertation we studied the generation of pattern-based pointcuts. We discussed inductive logic programming, a machine learning technique that can learn first-order logic rules from a collection of examples plus background information. We mapped the problem of finding a pattern-based pointcut to learning a logic rule. We have shown that given a collection of join points, a pattern-based pointcut can be learned.

8.2 Conclusion

Current aspect-oriented refactorings produce crosscuts based on enumeration of join points, which hinder further improvement of the "-ilities" in a program; in this thesis we demonstrate how machine learning techniques can learn pattern-based crosscuts.

In chapter 7 we have shown that pattern-based pointcuts can be generated from a set of join points. We created a logic representation of the static information about a Java program. We also modeled a join point graph that covers the possible join points graphs that can actually occur when executing the program. We identified some join points in this model as positive examples for a target pointcut. We gave these examples, along with the background information about the program, to the inductive logic programming algorithm FOIL. The result was a pattern-based pointcut that was suitable for use in a logic pointcut language.

8.3 Technical Contributions

To learn the pattern-based pointcuts from a collection of join points, we implemented FOIL, an inductive logic programming algorithm. The FOIL algorithm was developed in Smalltalk and uses SOUL as a logic engine. We implemented an interface for this algorithm that can read Java source files and generates logic facts. This interface depends on Irish, an extension to SOUL for Java reasoning, to parse and reason about the Java programs.

We validated the FOIL algorithm by performing experiments that show the generation of pattern-based pointcuts. The generated pointcuts can be used with CARMA for Java, our adaptation of CARMA that works on Java source.

8.4 Future Work

We present some topics that are based on this dissertation's work that may be interesting for future research or implementation.

- **Improving candidate selection** As we already discussed in the previous section, some candidates may have the same gain value. Adding a weight to some predicates would result in candidates having a higher gain value when they are composed from these predicates. This technique can guide the search and will result in better rules from the algorithm. We could also change the $Foil_Gain$ performance measure to take more information into account. It would be interesting to support preferences in the calculation, e.g. based on certain constants (instance variable names, *etc*), more control on the variables (preferring rules with less variables, or more).

- **Reducing noise** Some resulting rules may contain some noise. A possible automated solution would be to apply a heuristic function to detect bad smells in the rules. The results from this function can then be returned to the user, who can use this extra information to manually adapt the rule. Common bad smells could be handled by other tool support to enhance the rules.

- **Larger programs** The algorithm suffers from scalability problems. When more background information is available, much more candidates will be generated. The gain value has to be calculated for each candidate, and the gain function must also consider more options.

- **Comparing to other ILP algorithms** We only considered the FOIL algorithm in our experiments. It would be interesting to see how it compares to rules generated by other ILP algorithms, in particular RLGG.

- **Optimize induction algorithm** The current implementation is a very naive one, that can be improved on many places. Many optimizations can be incorporated to improve the speed of the induction. The implementation uses

SOUL as the logic engine and this is quite slow. A new implementation using a faster logic engine will improve the poor speed results.

- **Support for other languages** The current implementation works on Java programs. To support a new language, a parser has to be written that creates logic facts from the program and adds it to the background theory. A join point graph has to be constructed and a logic representation of it has to be added to the background theory. Summarized, to support another language, an "importer" has to written for it. As the result of the algorithm are logic rules, a logic pointcut language has to exist that allows crosscutting programs written in the new language.

- **Automated refactoring tool** An extension to an IDE could help developers refactor code. The extension should allow identification of the join points that the target pointcut should cover. The extension will feed the background information to the algorithm and start the induction. The learned rule can be shown in the IDE to let the developer verify its correctness.

Bibliography

[1] AspectJ website. URL `http://eclipse.org/aspectj/`.

[2] Frost webpage. URL `http://www.cincomsmalltalk.com/CincomSmalltalkWiki/Frost+Page`.

[3] Irish website. URL `http://prog.vub.ac.be/~jfabry/irish/`.

[4] CamelCase Wikipedia article. URL `http://en.wikipedia.org/wiki/CamelCase`.

[5] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, Cambridge, MA, 1996.

[6] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters, Feb. 21 2000.

[7] K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[8] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] R. Filman. Injecting ilities. In *Proceedings of the Aspect-Oriented Programming workshop at ICSE'98*, 1998.

[11] P. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley and Sons, 1994.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999. ISBN 0-201-48567-2.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1995. ISBN 0-201-63361-2.

[14] K. Gybels. Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure. Licentiate's thesis, Vrije Universiteit Brussel, Aug. 27 2001.

[15] K. Gybels and A. Kellens. An experiment in using inductive logic programming to uncover pointcuts. In K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, Sept. 2004.

[16] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual international Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, 2003.

[17] A. Kellens. Using inductive logic programming to derive software views. Licentiate's thesis, Vrije Universiteit Brussel, June 2003.

[18] A. Kellens and K. Gybels. Issues in performing and automating the "extract method calls" refactoring. In *Software-Engineering Properties of Languages and Aspect Technologies*, 2005.

[19] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001. ISSN 0001-0782.

[22] R. Laddad. Aspect-oriented refactoring series, Dec. 2003. URL `http://www.theserverside.com/`.

[23] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[24] M. P. Monteiro. Catalogue of refactorings for aspectj. Technical Report UM-DI-GECSD-200401, Universidade Do Minho, 2004.

[25] M. P. Monteiro and J. Fernandes. Object-to-aspect refactorings for feature extraction. In *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Mar. 2004.

[26] S. Muggleton and C. Feng. Efficient induction of logic programs. In *First Conference on Algorithmic Learning Theory*, 1990.

[27] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[28] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[29] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

[30] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, 2000. URL `http://www.research.ibm.com/hyperspace/`.

[31] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering (ICSE '99)*, pages 107–119, New York, NY, 1999. ACM.

[32] B. Tekinerdogan and M. Aksit. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, June 17 1998. URL `http://trese.cs.utwente.nl/aop-ecoop98/papers/Aksit.pdf`.

[33] T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views, Sept. 17 2003.

[34] T. Tourwé, A. Kellens, W. Vanderperren, and F. Vannieuwenhuyse. Inductively generated pointcuts to support refactoring to aspects. In *Workshop on Software Engineering Properties of Languages for Aspect Technologies*, 2004.

[35] F. Vannieuwenhuyse. Aspect-oriented refactoring. Licentiate's thesis, Vrije Universiteit Brussel, June 2004.

[36] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.