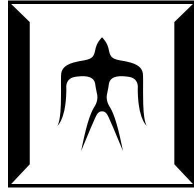
T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	関心事の分離のための破壊的拡張の表現力と安全性の向上
Title(English)	Expressive and safe destructive extensions for separation of concerns
著者(和文)	赤井駿平
Author(English)	Shumpei Akai
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第9053号, 授与年月日:2013年3月26日, 学位の種別:課程博士, 審査員:渡辺 治
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第9053号, Conferred date:2013/3/26, Degree Type:Course doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Science Tokyo Research Repository)



A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

---

Expressive and Safe Destructive Extensions for Separation of Concerns

関心事の分離のための破壊的拡張の表現力と安全性の向上

Shumpei Akai

Dissertation Chair:
Osamu Watanabe

February 2013,
Copyright © 2013 Shumpei Akai.
All Rights Reserved.

Abstract

Replacing or switching programs according to situations is important to develop software. It helps to get more performance, to enable platform-specific features, or to customize existing libraries. In C or C++, macros including #if or #ifdef are use to replace programs. However, they do not have enough modularity. Destructive extensions, including aspect-oriented programming and class extensions, are significant techniques in order to replace programs in modular way. They help to write a crosscutting concern in a module or a separated file. Pointcut/advice mechanism in aspect-oriented programming allows programmers to change behavior of expressions or method bodies in a program without modifying existing source code. Class extensions allow programmers to add or replace methods in existing classes.

Destructive extensions are used in several programming languages. However, three problems with destructive extensions have been discussed: improving expressiveness, fragile pointcut problem in AOP, and interference or conflict among extensions. More expressiveness encourages separation of more concerns including synchronization. Fragile pointcut problem causes failure to apply extensions. Interference among extensions makes it difficult to use multiple extensions together. These problems should be addressed in order that more programming languages adopt destructive extensions.

This thesis proposes three techniques to address these three problems. *Regioncut* is a language construct to treat arbitrary code regions as join points in an aspect-oriented programming language. Programmers can change behavior of code regions by using aspects with regioncuts. It improves expressiveness of aspect-oriented programming and enables to separate synchronization concerns into aspects. This thesis also proposes *assertion for advice* in order to address fragile pointcut problem in aspect-oriented programming.

It checks whether advice is applied to anticipated join points. This thesis proposes *method shelters* to avoid interference among class extension. They provide two types of scope of class extensions. Programmers can avoid interference by composing these scopes.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Professor Shigeru Chiba. He gave me great advice on studying, giving presentations, and writing papers. I also thank to the thesis committees: Professor Osamu Watanabe, Professor Masataka Sassa, Professor Shigeru Chiba, Professor Ken Wakita, and Professor Kazuyuki Shudo.

I deeply thank to the current and graduated members of Chiba Shigeru Group. In particular, I am deeply grateful to Masayuki Ioki, Fuminobu Takeyama, and Yung-Yu Zhuang.

Finally, I would like to thank mother, father, sister, and grandmother. This work would not have been possible without their moral and financial support.

Shumpei Akai
February 2013

Contents

1 Introduction **1**

1.1 Motivating Problem 2
1.2 Approach by This Thesis 4
1.3 Position of this thesis 6
1.4 Structure of this thesis 8

2 Program Replacement and Destructive Extensions **11**

2.1 Examples of Program Replacement 12
2.1.1 Changing Look and Feel of GUI Libraries 12
2.1.2 Switching Granularity of Synchronization 12
2.2 Destructive Extensions 14
2.2.1 Class Extensions 16
2.2.1.1 Smalltalk 16
2.2.1.2 Objective-C 16
2.2.1.3 Ruby 17
2.2.1.4 Usage of class extensions 17
2.2.2 Aspect-oriented programming and pointcut/advice mechanism 19
2.2.2.1 AspectJ 19
2.2.2.2 GluonJ 22
2.3 Problems with Destructive Extensions 23
2.3.1 Expressiveness 23
2.3.1.1 Separation of synchronization concerns in AOP . 26
2.3.1.2 Separation of exception handling concerns . . . 28
2.3.2 Fragile Pointcut Problem 29

2.3.2.1 Fragile Pointcut Problem with synchronization concerns 31
2.3.3 Conflict among Destructive Extensions 31
2.3.3.1 Scopes of class extensions 34
2.4 Techniques Similar to Destructive Extensions 35
2.4.1 Java class loader 35
2.4.2 C# 36
2.4.3 Virtual classes 37
2.4.4 Context-oriented programming 37
2.4.5 Implicit conversion in Scala 38
2.4.6 Type classes 39
2.5 Summary 42

3 Regioncut 43
3.1 Overview 43
3.2 Semantics 45
3.2.1 Matching between regioncuts and a sequence of statements 46
3.3 Context Exposure 50
3.4 Implementation Issues 50
3.4.1 Analysis of blocks and statements 50
3.4.2 Around advice support 51
3.5 Evaluation 58
3.5.1 Javassist 58
3.5.2 Hadoop 58
3.6 Current Limitation 60
3.7 Fragility of regioncuts 67
3.8 Summary 68

4 An Assertion for Advice 71
4.1 Overview 71
4.2 Implementation 74
4.3 Evaluation 74
4.4 Summary 75

5 Method Shelters 77
5.1 Overview 77
5.2 Lookup semantics 80
5.2.1 Method shelter tree 80
5.2.2 The lookup algorithm 81

5.3 A proof-of-concept implementation 88
5.3.1 Implementation details 88
5.3.2 Optimization 89
5.3.3 Compatibility 90
5.4 Applications 91
5.4.1 Convenient methods in Ruby on Rails 91
5.4.2 Operator redefinition 93
5.4.3 RSpec 93
5.4.4 Protecting optimized methods 94
5.4.5 Private instance variables 94
5.5 Performance 99
5.5.1 Micro benchmark 99
5.5.2 tDiary 100
5.5.3 Ruby on Rails 101
5.6 Summary 102

6 Conclusion **105**

Bibliography **109**

List of Figures

1.1 Trade-off between expressiveness and fragility in programming languages. 7
1.2 Trade-off between expressiveness and interference among class extensions in programming languages. 8

2.1 Different look and feel of AWT and Swing 13
2.2 Swing's JButton class is not a subclass of AWT's Button class. . 14
2.3 Javassist's ProxyFactory class have two candidates for synchronization. 15
2.4 Example of Class Extensions in Smalltalk 16
2.5 Example of Class Extensions in Objective-C 17
2.6 Example of Class Extensions in Ruby 17
2.7 Methods that access a database 21
2.8 Example of Logging aspects in the Around Advice 22
2.9 Example of a Reviser in GluonJ 22
2.10 Example of Tracematch 24
2.11 Example of LoopsAJ 25
2.12 Example of loop with several successor nodes 25
2.13 Example of Synchronized Block Join Point 26
2.14 Example of EJFlow 27
2.15 Pseudo code of pointcut to synchronize a code 28
2.16 A region that includes file operations may throw exceptions . . . 29
2.17 Example of *open modules* 30
2.18 A problem in classboxes 33
2.19 Example of Ruby's *Refinements* 34
2.20 Example of a definition of an extension method 36

2.21 Example of an extension method call 37
2.22 Example of virtual classes in Newspeak 38
2.23 Example of context-oriented programming in ContextJ 39
2.24 Example of implicit conversion in Scala 40
2.25 Example of type classes in JavaGI 41

3.1 A regioncut designator 44
3.2 A regioncut designator can have three or more pointcuts. 44
3.3 A method including two similar regions 45
3.4 eval-method function for evaluating a program with reginocut . . 47
3.5 eval-stmt function for evaluating a statement with reginocut . . . 48
3.6 Range and order of evaluation with reginocut 48
3.7 Matching function match for regioncut 49
3.8 Context exposure by a regioncut 51
3.9 Example of *markers* 54
3.10 A local variable is updated within a code region 55
3.11 A transformed version of the method in Figure 3.10 55
3.12 A program including jumps to out of a selected code region . . . 56
3.13 A transformed version of the method in Figure 3.12 57
3.14 Fine-grained Synchronization by hand 61
3.15 Coarse-grained Synchronization by hand 62
3.16 The advices for fine-grained synchronization 62
3.17 The advice for coarse-grained synchronization 63
3.18 Example Code from Hadoop and Its Synchronization Aspect 1 . . 64
3.19 Example Code from Hadoop and Its Synchronization Aspect 2 . . 65
3.20 Example of conflict of regioncuts 66
3.21 A region before/after removing a method call 67
3.22 A region before/after reordering a method call 68
3.23 An advice that handles an exception thrown from Figure 2.16 . . 69

4.1 An annotation for a method 72
4.2 An annotation for an advice 72
4.3 Advice with @SolveProblem is invoked by method with @AssertAdvised 73
4.4 Method with @AssertAdvised is invoked by advice with @SolveProblem 74

5.1 Code sample (a solution of the problem in Figure 2.18) 83
5.2 Ambiguous methods in a method shelter 84
5.3 An example of an import graph of method shelters 84

5.4 A method shelter tree reconstructed from Figure 5.3 84
5.5 Method lookup functions of method shelters 85
5.6 Definition of source-node and source-chamber 85
5.7 An example of source-node and source-chamber 86
5.8 Definition of lookup-exposed and lookup-hidden 87
5.9 The syntax of method shelters 89
5.10 The definition of Ruby's stack frame struct. 90
5.11 The definition of Ruby's inline cache struct. 91
5.12 The time-related methods we defined in a method shelter 92
5.13 A client code of Ruby on Rails 92
5.14 The code that redefines "/" methods in method shelters 96
5.15 "avg.rb" library that redefines / and adds average method to Array class 97
5.16 "avgspec.rb" spec file for average method written in RSpec . . . 97
5.17 "divspec.rb" for Fixnum's / written in RSpec 97
5.18 The code for defining getter and setter methods to access a private instance variable 98
5.19 The client code using accessor methods to a private instance variable 98
5.20 The benchmark program that calls an empty method under five method shelters 100
5.21 The benchmark program for Ruby on Rail 102

List of Tables

2.1 Typical Pointcuts in AspectJ 20
2.2 Summary of existing mechanisms. 42

3.1 The execution time of the Javassist benchmark 59
3.2 The number of synchronization concerns in the TaskTracker class 60

4.1 The number of synchronization advices with regioncuts after the update to Hadoop 0.18.3 75

5.1 Execution time of empty method (1,000 tries) 99
5.2 Execution time of fib(33) (1,000 trials) 100
5.3 Response time of tDiary (300 trials) 101
5.4 Response time of Rails application (1,000 trials, development env.) 102
5.5 Response time of Rails application (1,000 trials, production env.) 102
5.6 Cache hit ratios of Rails application (1,000 trials, production env.) 103

Chapter 1

Introduction

The concept of *separation of concerns* — or decomposing programs into natural units rather than units imposed by tools — is a significant idea for developing software [44, 45]. It helps divide complicated problems into simpler subproblems. Responsibilities of subproblems will get clear and minimum by separation of concern. The idea is important to develop large and complex software systems.

Abstraction is one aspect of separation of concerns. Many abstraction techniques appeared to realize separation of concerns. Procedures, subroutines, functions, or methods are atomic units that represent subprograms in high-level programming languages.

Several module systems are proposed to *modularize* subprograms. *Object-oriented programming* introduced *objects* to couple and abstract procedures and data structures. Common object-oriented programming languages also introduced *classes*, which is a language construct for defining an object's structure and behavior. Some programming languages also provide modules that contain procedures and classes, for example, *namespace* in C++ or *package* in Java.

Modularization increases changeability of programs. Loosely-coupled modules get interfaces between other modules. Modules communicate through defined interfaces. Modules can be easily replaced and it enables

to change behavior of programs without modifying other code. Loosely-coupled modules can be developed independently. Modularization also increases comprehensibility of programs. Appropriately designed modules are less dependent on other modules. Ideally, you can understand a module by knowing its input and output.

A number of programming paradigms or modularization techniques are proposed for achieving better modularization. Aspect-oriented programming [35] (AOP), subject-oriented programming [27], feature-oriented programming [11], and context-oriented programming [30] are the programming paradigms to improve or complement object-oriented programming. Class extensions, virtual classes [38, 40], and types classes [56] introduce more flexible module systems.

Aspect-oriented programming and class extensions can destructively change behavior of existing programs without editing the base code. This thesis defines this feature as *destructive extensions*. Destructive extensions are useful for separation of *crosscutting concerns*. Crosscutting concerns are concerns that crosscut inheritance hierarchies. Modularizing crosscutting concerns are not suitable for object-oriented programming. Aspect-oriented programming has a feature to add or replace behavior to specified execution points — method call, field access, etc. Crosscutting concerns including logging and error handling can be separated into an *aspect* by aspect-oriented programming. Class extensions allow programmers to add or redefine methods in existing classes. You can reuse and customize complicated libraries without modifying their source code by using class extensions. Class extensions enable to add several utility methods into existing classes, and the addition can be placed in one module.

1.1 Motivating Problem

To replace or switch programs is important for developing software. For example, operating systems, interpreters, or compilers contain platform-dependent code. To switch such a platform-dependent code, #if or #ifdef macros are used in C or C++.

Synchronization code may also be platform-dependent. Granularity of synchronization may affect performance of multi-threaded programs. Since fine-grained synchronization increases concurrency, it improves performance on a computer that has many CPU cores. However, fine-grained synchronization increases overhead of synchronization. Coarse-grained synchronization

improves performance on a computer that has a few CPU core since it decrease overhead of synchronization. Switching granularity of synchronization code according to the machine is important to get better performance.

Replacement of programs is also useful to reuse and customize existing libraries or programs. Suppose that changing look and feel of GUI library developed in an object-oriented programming language. You can create a new GUI library by subclassing the classes in the existing library even if you cannot directly edit that source code. For example, Swing, which is a GUI library for Java, is developed with subclassing AWT's classes. However, it introduces many scattered codes, which change look and feel in the classes. Moreover, the class hierarchies of AWT and Swing do not match since Java does not allow multiple inheritance. Duplicated codes were needed to implement Swing. If you can replace only look-and-feel-related code in the existing GUI library, a new GUI library with different look and feel can be easily created. This example shows us that customizing existing libraries is also crosscutting concerns, and it should be separated into modules.

Language constructs to replace programs should be modular. Macros in C or C++ are traditionally used to replace or switch programs according to situations. However, they do not help to make code modular. Object-oriented programming allows us to reuse and customize existing classes by subclassing. However, in order to customize existing libraries by creating subclasses, you should modify all instantiation code in the library to make instances of the subclasses. You have to modify the existing library directly to replace programs even in object-oriented programming languages. Destructive extensions including aspect-oriented programming and class extensions help to replace programs in modular way.

Several problems with destructive extensions are discussed in communities of aspect-oriented programming and languages which have class extensions: expressiveness, fragile pointcut problem, and interference among extensions. To replace programs in a modular and practical way, these problems should be solved or relieved.

In order to separate more concerns into modules, destructive extensions should be more expressive. For example, synchronization concerns are not suitable for aspect-oriented programming. In AspectJ [34], which is an AOP language, you can change behavior of *join points*: a method call, method execution, or field access. However, regions that should be synchronized may not take a form of these join points. AOP languages should have means to select more flexible join points to deal with synchronization that has several granularity.

Expressiveness of destructive extensions may decrease safeness. Aspect-oriented programming got *fragile pointcut problem* [51] in exchange for its expressiveness. AOP systems do not emit warning when an advice is not correctly applied even if the advice must be applied to work correctly. Destructive extension should provide a way to guarantee that at least one extension is applied.

Class extensions improve expressiveness of object-oriented programming. They introduce a risk of interferences or conflicts among extensions. Conflicts among class extensions break programs. Programs may fail to be compiled or work incorrectly. Destructive extensions should also provide a way to control multiple extensions applied to one place.

Systems for modularization should be improved to solve these problems. Aspect-oriented programming and class extensions improved expressiveness of programming languages. However they have room for improvement in expressiveness. They should also have means to solve fragile poinctut problem and interferences in order to write programs more safely. This thesis tackles these three problems by introducing programming language constructs.

1.2 Approach by This Thesis

Section 1.1 described three problems with destructive extension: *expressiveness*, *fragile pointcut problem*, and *interference*. This thesis proposes three language constructs of programming languages to address these problems with destructive extensions: *regioncut* to handle synchronization concerns in AOP languages, *assertion for advice* to check whether pointcuts or regioncuts are correctly applied, and *method shelters* to avoid conflicts among class extensions. Regioncuts improve expressiveness AOP, and assertion for advice make AOP less fragile, and method shelters decrease a risk of interferences among class extensions.

Regioncut

This thesis proposes *regioncut* [5, 3] to separate synchronization concerns into aspects. Regioncut is an extension of AspectJ to treat code regions as ordinary join points. An ordinary join point in AspectJ is a single expression or a method body. A code region is a sequence of statements. Regioncut provides means to select code regions that you need and to change behavior of the selected region. You can select an arbitrary code region by specifying

(sub)sequence of join points that the region contains. Regioncut does not select a code region that partially overlaps with control structures. Hence, a compiled code with regioncuts does not emit illegal byte code sequences.

Regioncut enables us to select code regions and to apply around advice to the regions. You can synchronize a critical section by writing synchronization code in an around advice. Synchronization concerns can be separated into aspects. You can switch granularity of synchronization by switching aspects.

Regioncut allows us to write synchronization with several sizes of granularity in aspects. It is difficult to do in existing aspect-oriented programming languages. Regioncut improves expressiveness of AOP languages.

Assertion for Advice

Aspect-oriented programming introduced fragility of programs. Regioncut improved expressiveness of AOP; therefore AOP may get more fragile. Synchronization code must work correctly at run time. However, synchronization written in an aspect may be fragile. If a base code, which needs synchronization, is refactored, the synchronization aspect may fail to be applied to the code. Because of obliviousness property [23] of AOP, a compiler tells us nothing even if an aspect is applied incorrectly.

To address fragile pointcut problem in AOP, this thesis proposes an *assertion for advice* [3]. To use assertion for advice, you should attach annotations to methods and advice. Annotations of assertion for advice take a name of a concern. If the advice annotated with the concern name A may call the method annotated with A, the proposed system does not emit warning. On the other hand, if the method may call the advice, it does not emit warning either. Otherwise, warning is emitted. With assertion for advice, you can easily find aspects that do not work.

Method Shelters

Class extensions improved expressiveness of object-oriented programming languages. However, class extensions caused a risk of conflicts of method definitions. In order to avoid such a conflict, means to restrict a scope of class extensions is required. Several module systems were proposed to introduce a scope of class extensions. However, they do not provide fine-grained scope.

This thesis proposes *method shelters* [4] to control a scope of class extensions and address conflicts. A method shelter is a module that confines

effect of class extensions. A method shelter can *import* other method shelters to call or override methods in them. To control a scope flexibly, a method shelter consists of two parts: *exposed chamber* and *hidden chamber*. Methods defined in an exposed chamber can be called and overridden by other method shelters. Methods defined in a hidden chamber are not visible from other method shelters which are importing that shelter.

1.3 Position of this thesis

The position of this thesis in the research history of programming languages is that it provides mechanisms to elevate expressiveness of programming languages without spoiling safeness including fragility and interference. As illustrated in Figure 1.1, expressiveness and fragility in programming languages are trade-off. Object-oriented programming languages are more expressive than procedural languages. However, in object-oriented languages, modifying superclass may affect subclasses. Aspect-oriented programming introduced more expressiveness than object-oriented programming; it caused fragile pointcut problem. However, aspect-oriented programming languages are not enough expressive. For example, it is difficult separate synchronization concerns into modules using aspect-oriented programming. This thesis presents regioncut to make aspect-oriented programming more expressive. Regioncut improved expressiveness of aspect-oriented programming by introducing means to treat code regions as join points. Granularity of traditional join points is not flexible since they are expressions or method bodies. These join points are not suitable for synchronization, exception handling, and resource handling. Join points with more flexible granularity, code regions, introduced ability to separate more concerns into module. However, introduction of expressiveness by regioncut also make aspect-oriented programming more fragile. This thesis also proposes assertion for advice to address trade-off between expressiveness and fragility introduced by regioncut. Assertion for advice provides means to guarantee that an advice is applied to an anticipated point by annotating a method and an advice. Existing aspect-oriented programming languages do not provide a way to show whether an advice is applied or not. The compiler proposed by this thesis checks whether an advice affects the specified methods at compile time. Combination of regioncut and assertion for advice addresses trade-off between expressiveness and fragility in aspect-oriented programming languages.

As illustrated in Figure 1.2, expressiveness and a risk of interference

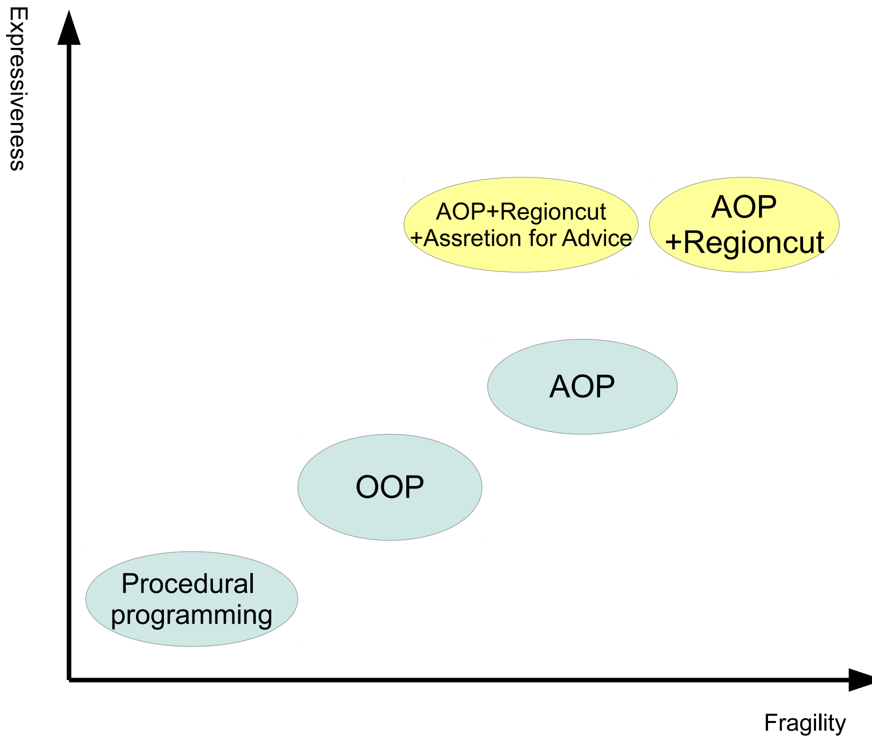


Figure 1.1. Trade-off between expressiveness and fragility in programming languages.

among extensions are also trade-off. Class extensions introduced more expressiveness into object-oriented programming. On the other hand, they introduced a risk of interference. Exiting approaches including Ruby's refinements and classboxes tried to decrease the risk. Refinements restrict the scope of class extensions in lexical scope. It solves a risk of interference. However, it spoils expressiveness of class extensions. Classboxes are safer than normal class extensions and more expressive than refinements. However, there are several cases that programmers cannot avoid interference using classboxes. This thesis presents method shelters to avoid a risk of interference introduced by class extensions. Method shelters are more expressive than refinements since they can replace method in existing libraries. Method shelters provide means to avoid interference that classboxes cannot solve. To make class extensions expressive without interference, method shelters provide two scopes of class extensions. Programmers can avoid interference

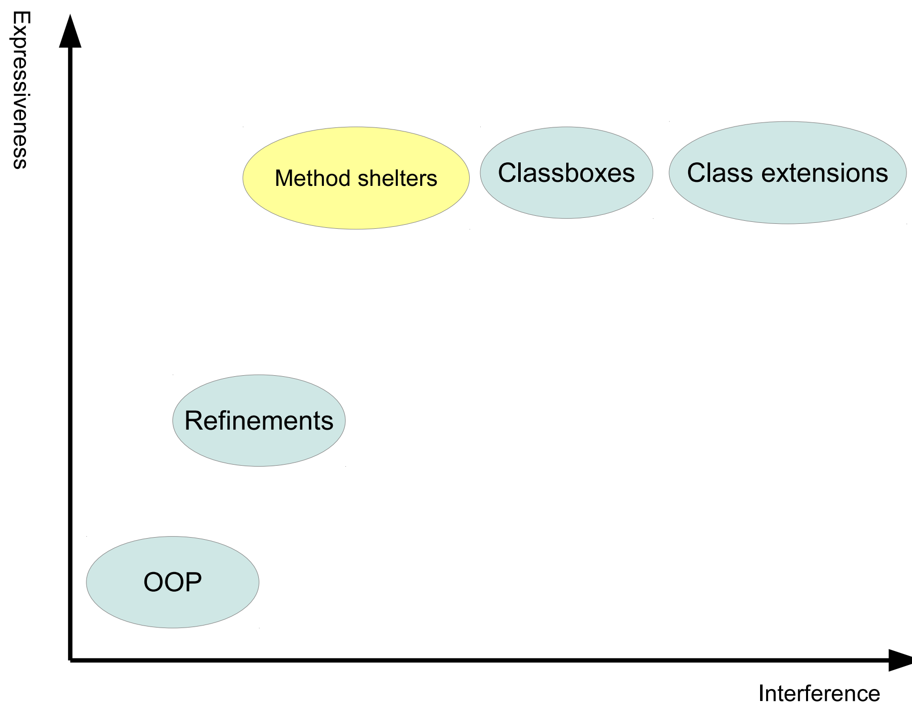


Figure 1.2. Trade-off between expressiveness and interference among class extensions in programming languages.

among class extensions by composing these scopes and modules.

1.4 Structure of this thesis

From the next chapter, this thesis explains background of this thesis and details of the proposed language constructs. The structure of the rest of this thesis is as follows:

Chapter 2: Program Replacement and Destructive Extensions

This chapter discusses needs of destructive extensions and shows that destructive extensions help us to replace programs according to situations. This chapter also explains problems with existing destructive extensions tech-

niques: expressiveness, fragile pointcut problem, and interference among extensions.

Chapter 3: Regioncut

To improve expressiveness of AOP and support synchronization concerns in AOP, this chapter introduces a new language contruct named *regioncut*. This chapter explains design, semantics, implementation issues, and application to existing programs.

Chapter 4: An Assertion for Advices

To guarantee that a mandatory advice is correctly applied and address fragile pointcut problem in AOP, this chapter presents *assertion for advice*. This chapter explains design, implementation, and evaluation of assertion for advice.

Chapter 5: Method Shelters

This chapter introduces *method shelters*. It provides means to control scope of class extensions and address interferences or conflicts among class extensions. This chapter describes design, semantics, and implementation of method shelters. Then, this chapter also explains application to several programs and performance measurement of method shelters.

Chapter 2

Program Replacement and Destructive Extensions

It is important to replace programs according to situations for developing large software. For example, operating systems have codes that are dependent on hardware. Compilers or interpreters contain codes that are dependent on operating systems and hardware. In order to support threads in programming languages on multiple platforms, a layer, which wraps different thread libraries provided on operating systems, is required, and its codes should be replaced according to operating systems. Using asynchronous I/O is also platform-dependent. To implement server software which handles many TCP connections, asynchronous I/O should be used. However, operating systems do not have standard API for asynchronous I/O: Linux provides *epoll* and BSDs provide *kqueue*. As this example shows, replacing programs is also required for applications to get better performance.

To replace or switch such a code, macros — #if and #ifdef — are used in C or C++. You can enable or disable platform-dependent programs by defining constants using #define. Macros in C or C++ do not have enough modularity. #if and #ifdef are placed directly in the source code. Replacement code may be scattered in several places. To replace existing program, a modifying code should placed in a module, and it should modifies the existing code without editing the base code. This chapter describes importance

of program replacement, techniques for modifying existing programs named *destructive extensions*, and problems with destructive extensions.

2.1 Examples of Program Replacement

2.1.1 Changing Look and Feel of GUI Libraries

GUI libraries are another example. A GUI library may be customized to make modern look and feel or make suitable for a platform. To change look and feel, macros can be used in C. However, the code for customizing is tangled with the base code, and it is not modular. Swing library realized changing look and feel by inheritance.

Swing is a GUI library for Java, which is based on AWT. Swing changes look and feel from AWT (Figure 2.1). Since Java does not have means to modify existing classes, Swing is developed by creating subclasses of classes in AWT.

In AWT, Component class is the root class of GUI widget classes. Other widget classes including Button class are subclasses of Component. In Swing world, JComponent is the root class and it is a subclass of Component class. Swing's widget classes must inherit from JComponent. Since multiple inheritance is not allowed in Java, Swing's widget classes cannot inherit from AWT's classes. JButton class is a subclass of JComponent instead of Button class (shown in Figure 2.2). Although both Button and JButton implement button widget, JButton cannot reuse Button class's code. In Java, duplicated code is needed to make customized library.

It is difficult to customize existing library without code duplication. Customizing codes are appeared in various places and they are crosscutting concerns. You can change look and feel of a GUI library by redefining methods of existing classes if a programming language supports redefinition of existing classes without modifying their source code.

2.1.2 Switching Granularity of Synchronization

In 2006, the developers of Javassist [17] received a bug report [1]. Javassist is a Java class library for modifying Java bytecode and it is widely used in a number of Java products, mainly web application frameworks such as Redhat JBoss application server and Hibernate. The bug was that a method

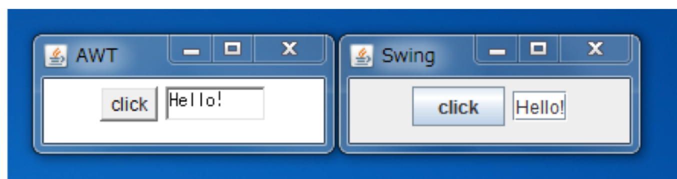


Figure 2.1. Different look and feel of AWT and Swing

generating a proxy object was not thread-safe: there is no synchronized statement. To fix this bug, they had to modify the method to contain synchronized statements.

An interesting issue of this bug fix was lock granularity; which code block should be put into a synchronized statement. Since lock granularity affects concurrency, minimizing the granularity generally improves execution performance when multiple processor cores are available. In this case the developer have two candidates of synchronization granularity (shown in Figure 2.3). The blue box means coarse-grained lock, and the two red boxes mean fine-grained lock.

However, as discussed in [37], excessive concurrency often have negative impact on performance. In year 2006, low-end servers were still single-processor machines and 4-way multi-processors machines were expensive (Intel Core-MA Xeon "Woodcrest" was shipped in 2006). On a single- or 2-way machine, small granularity may not improve execution performance under a heavy workload. In programs with small granularity synchronization invoke lock and unlock operations many times, and these operations spend more time than other operations. Small granularity synchronization may reduce performance. Thus, for the users who run their software on such a relatively slow machine, we should have modified Javassist to make the lock granularity larger (the blue lock in Figure 2.3).

This experience shows that a synchronization concern is a good candidate to replace programs according to situations. If multiple implementations of the synchronization are supplied as a set of modules, users can choose the best implementation and apply the module for that implementation when they install the software. They do not have to modify the rest of the program when they change the implementation. On the other hand, when the developers of Javassist fixed the synchronization bug, they had to choose one implementation and hard-wire it since the software was written in pure Java. The resulting software ran fast on some kind of hardware but not on other

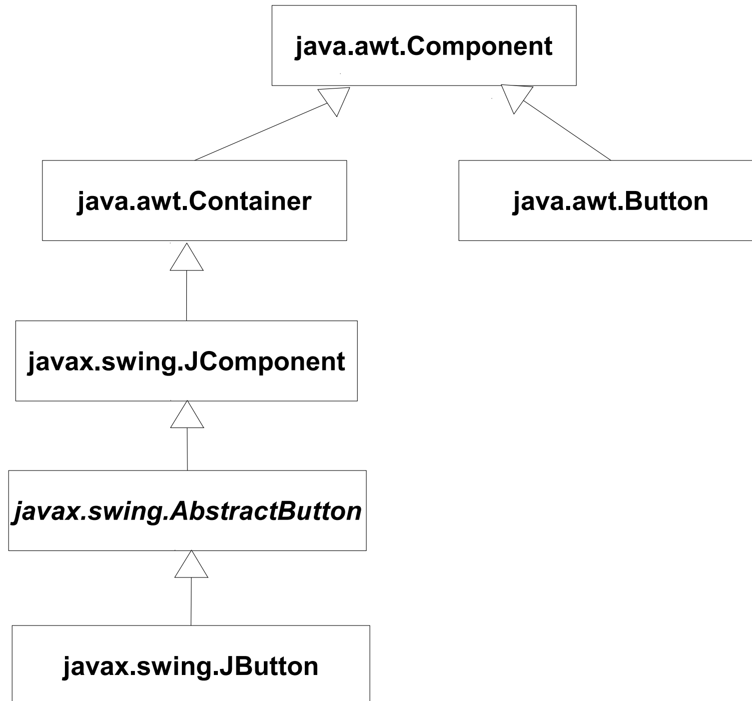


Figure 2.2. Swing's JButton class is not a subclass of AWT's Button class.

kinds.

2.2 Destructive Extensions

Modification of existing programs' behavior is used in several programming languages. We call the features that can change existing programs without modification of the base code as *destructive extensions*. Destructive extensions include two features: pointcut/advice mechanism in *aspect-oriented programming* and *class extensions*. Smalltalk, Objective-C, and Ruby have class extensions. Poincut/advice mechanism in aspect-oriented programming languages is means to modify behavior of expressions or method bodies in a program.

Destructive extensions improve modularity of programs in object-oriented programming. In most OO-languages, programs are packed in *classes*. A class includes definition of data structure and operations related to the data structure. Programs related to a data structure can be put in one place.

```
public class ProxyFactory {
  public Class createClass() {
    if (thisClass == null) {
      ClassLoader cl = getClassLoader();
      synchronized (proxyCache) {
        if (useCache){createClass2(cl);}
        else {createClass3(cl);}
      }}
    return thisClass;}
  private void createClass2(ClassLoader cl) {
    CacheKey key = new CacheKey(…);
    synchronized (proxyCache) {
      HashMap cacheForTheLoader =…;
      if (cacheForTheLoader == null) {
        cacheForTheLoader = new HashMap();
        proxyCache.put(cl, cacheForTheLoader);
        cacheForTheLoader.put(key, key);
      }else {…}}
    synchronized (key) {
      Class c = isValidEntry(key);
      if (c == null) {
        createClass3(cl);
        key.proxyClass = new WeakReference(…);
      }else{thisClass = c;}
    }}}
```

Figure 2.3. Javassist's ProxyFactory class have two candidates for synchronization.

Programmer can reuse code by inheritance.

If you only use OO features, you cannot write some code in modular way. It is difficult to modularize features or concerns that are spreading across many classes without inheritance. Logging is an example. Logging code may appear several places and crosscut class hierarchies. Whether logging code to be required or not depends on what the method is doing. You should write logging code directly in the method to implement logging feature.

In order to modularize such a features or concerns — *crosscutting concerns* —, existing classes should be destructively extensible from outside of the class definitions. The extension should be placed in one file or a module. Destructive extensions allow us to separate crosscutting concerns into a module. This section describes class extensions and aspect-oriented pro-

```
1 Array extend [
2     average [
3         ^(self inject: 0 into: [:r :i | r + i ]) / (self size)
4     ]
5 ]
```

Figure 2.4. Example of Class Extensions in Smalltalk

gramming.

2.2.1 Class Extensions

Destructive extensions in OOP languages are called *class extensions*. Several languages provide class extensions as languages' feature.

2.2.1.1 Smalltalk

Smalltalk [24] has class extensions feature. You can extend existing classes. Figure 2.4 shows an example of class extensions in Smalltalk. This example adds new average method into Array class. This syntax is same as normal class definitions.

2.2.1.2 Objective-C

Objective-C is a OOP programming language based on C. It introduces Smalltalk-style OOP into C. Objective-C can also extend existing classes. Class extensions feature is called *category* in Objective-C.

In Cocoa and Cocoa touch framework, which are core framework in Mac OS X and iOS, use category to extend classes in separated files. For example, Cocoa touch framework contains Foundation Kit library and it defines NSString class, which represents a string, and UIKit library adds several methods to NSString class. UIKit defines sizeWithFont: methods in NSString, which calculate size of the receiver string if it is rendered with the specified font. UIKit library adds methods to the existing class instead of defining new classes for user's convenience.

Figure 2.5 is an example of category or class extension in Objective-C. It adds *average* method to NSArray class, the array class in Cocoa. The difference of syntax between category and normal class definition are

```
1 @implementation NSArray(AverageAddtion)
2 - (double)average{
3     double sum=0;
4     for(NSNumber* num in self){
5         result+=[num doubleValue];
6     }
7     return sum/[self count];
8 }
9 @end
```

Figure 2.5. Example of Class Extensions in Objective-C

```
1 class Array
2   def average
3     self.inject(0){|r,i| r+i }.to_f / self.size
4   end
5 end
```

Figure 2.6. Example of Class Extensions in Ruby

"(AverageAddtion)" placed next to NSArray. If you destructively extend a class, you must specify explanation of the extension after the class name.

2.2.1.3 Ruby

Class extensions are called *open class* in Ruby [48]. You can always extend existing classes like Smalltalk. Figure 2.6 shows an example. Array class is the built-in array class and average method is added to the class. The syntax of class extensions in Ruby is the same to normal class definition.

2.2.1.4 Usage of class extensions

Class extensions are frequently used in Ruby. For example, a number of use cases are found in Ruby on Rails [46]. We below show typical usage of class extensions in Ruby.

Convenient methods Class extensions are used to add convenient methods to core classes: Integer, String, Array and so on. For example, in Ruby on Rails, a suite of bytes methods is added to Numeric class, which is a super class

of Integer and Float classes. The method call "n.kilobytes" (where n is a number) returns $n \times 1024$ and "n.megabytes" returns $n \times 1024^2$. These bytes methods are useful when writing a program that handles file sizes. Programmers' intentions will be clear.

Another example is sum method. Ruby on Rails also adds sum method to Enumerable module, which is a mixin [15] module for list-like classes. This method computes the sum of elements in an Enumerable object. This method is simple and useful although the Ruby's standard library does not provide it. By using class extensions, third party libraries such as Ruby on Rails can easily add convenient methods.

Operator redefinition In Ruby, several operators, such as +, -, * and /, are normal methods. Thus anyone can redefine them.

Division of integers in Ruby returns an integer by default, for example, 1/2 returns 0. On the other hand, the Ruby's standard library mathn redefines it. Once you load this library, division of integers returns a rational. 1/2 returns a Rational object that represents $\frac{1}{2}$. This library makes it possible to describe mathematical expressions with normal notations.

Tree traversal Class extensions simplify tree traversal. If you naively write traversal code separately from tree-node classes, the code includes runtime type checking and it must be modified when a new node class is added. Although the code following the Visitor pattern is more extensible, all node classes must have methods for the Visitor pattern in advance. The Visitor pattern is not applicable to a tree if the node classes do not conform the Visitor pattern or they are not modifiable since a third-party library provides them.

If class extensions are available, you can add methods for traversal to node classes on demand. For example, suppose that a tree consists of Integer and Array and you want to sum up every integer elements in a tree. You only have to write the following code

```
1 class Integer
2   def sum_tree
3     self
4   end
5 end
6
7 class Array
8   def sum_tree
9     result=0
```

```
10      for child in self
11        result += child.sum_tree
12      end
13      result
14    end
15  end
```

The Visitor pattern is not required to traverse trees if you have class extensions. The code including runtime type checking is not needed.

Serialization libraries for Ruby often use this technique. For example, a JSON [19] library for Ruby adds to_json method to core classes, such as Integer, String, Array and Hash. A JSON serializer uses this method to traverse a tree made by core classes and to dump a JSON file.

Monkey patching When a third-party library has a bug, class extensions allow programmers to patch and fix it. A method that includes a bug can be replaced with a correct implementation of that method. Programmers do not have to directly modify the source code of the library. This technique is known as *monkey patching*.

2.2.2 Aspect-oriented programming and pointcut/advice mechanism

Aspect-oriented programming (AOP) [35] is proposed as a technique to complement procedural and object-oriented programming. It allows programmers to modify programs from outside of their modules to improve modularity. AOP aims to modularize concerns including logging, error handling or synchronization, which are scattering in Object-Oriented Programming. In AOP, these concerns are called *crosscutting concerns*. AOP helps to separate crosscutting concerns into a new kind of module named an *aspect*.

AOP has two mechanisms: pointcut/advice and class extensions. Class extensions are mentioned in Section 2.2.1. This section describes AOP languages and pointcut/advice mechanism.

2.2.2.1 AspectJ

AspectJ [34] is a Java-based programming language that supports aspect-oriented programming. AspectJ introduce several concept and constructs

pointcuts	*join points the pointcut matches*
call(*method*)	when the *method* is called
execution(*method*)	when the *method* is executed
get(*field*)	when the *field* is read
set(*field*)	when the *field* is set
within(*class*)	join points which appear in the *class*
within(*method*)	join points which appear in the *method*
this(*var*)	when the *this* at join point is instance of *var*'s type
target(*var*)	when the receiver of join point is instance of *var*'s type
args($arg_0, arg_1, \ldots$)	when arg_n is instance of *var* 's type
cflow(*pointcut*)	when the join point in the control flow of the *pointcut*

Table 2.1. Typical Pointcuts in AspectJ

including *join points*, *pointcut-advice* and *inter-type declarations* for separation of crosscutting concerns. These allow programmers to modify behavior of programs from outside of a module.

Join points The concept of join points is introduced by AspectJ. A join point is an execution point of programs. For example method calls, executions of a method body, constructor calls and filed accesses are considered as join points in AspectJ.

Pointcuts A pointcut is a language construct for specifying *join points*. Programmers selects intended join points from all join points in the program. Pointcuts provided by AspectJ are showed in Table 2.1. Pointcuts can be composed by logical operator: conjunction and disjunction. For example, a pointcut

```
call(void Lock.lock()) && withincode(String WebController.action())
```

picks out calls of Lock.lock method in the body of WebController.action() method.

Advice Advice is a procedure to define additional behavior at join points specified by pointcuts. AspectJ provides three types of advice: *before*, *after* and *around*. Before advice adds an additional behavior before the join point, and after advice adds it after the join point. Around advice replace the behavior of the join point, and *proceed* special function allows you to call the original behavior of the join point.

```
1  class FooController{
2    Database db = ...;
3    void fooAction(){
4        //...
5
6        Result r = db.get(searchQuery);
7
8        //...
9    }
10
11   void barAction(){
12       //...
13
14       Result r = db.get(updateQuery);
15
16       //...
17   }
18 }
```

Figure 2.7. Methods that access a database

Aspects and pointcut/advice mechanism An aspect is a module that can include one or more advice. Programmer can change behavior of other modules from an aspect, which is outside of the modules.

Figure 2.7 shows methods that access a database. If you need to log accesses occurred in the methods, pointcut/advice can be used. Figure 2.8 shows an example of a logging aspect. The call pointcut in the advice selects calls of Database.get methods in the methods. When execution of the program reached the call of Database.get, the advice will be executed. The proceed call in the advice will execute the selected join point.

Aspects can be applied before executing programs. In AspectJ, advice code of aspects is inserted into base code at compile-time or load-time. This insertion is called *weaving*. Weaving introduces better performance of aspects execution.

Inter-type declarations Class extensions are called as *inter-type declarations* in AspectJ. Aspects in AspectJ have ability to define fields, methods and constructors of other classes. Aspects can modularize concerns or features that do not adapt to class hierarchies by using inter-type declarations.

```
1 aspect Log{
2         void around() :
3           call(Result Database.get(String))
4         {
5           Logger.log("access started");
6           proceed();
7           Logger.log("access finished");
8         }
9 }
```

Figure 2.8. Example of Logging aspects in the Around Advice

```
1 class AddLog revises Database{
2         Result Databaes.get(String query) {
3           Logger.log("access started");
4           super.get();
5           Logger.log("access finished");
6         }
7 }
```

Figure 2.9. Example of a Reviser in GluonJ

2.2.2.2 GluonJ

GluonJ [18] is another aspect-oriented programming language but it does not have AspectJ-like constructs. GluonJ provides AOP features by enhancing OOP mechanisms. The enhancement includes *revisers* and *within predicates*.

Revisers A reviser is familiar to a class definition, but it destructively adds or replaces target class's member. GluonJ provides revisers to simulate and around advice execution pointcut. Figure 2.9 shows an example of a reviser. This example contains only one GluonJ-specific construct: *"revises"*. The super call in the reviser method works as proceed() in AspectJ. The users of GluonJ should know less special constructs than existing AOP languages.

Within predicate Scope of simple destructive method revising is global. GluonJ introduces predicate dispatch with within predicate to limit the scope of reviser. A within predicate takes a class or method name as an argument. A method defined with within argument is executed if the method is called

from the specified location. With within predicates, GluonJ has capability of the call pointcut with within/withincode pointcut in AspectJ.

2.3 Problems with Destructive Extensions

The three significant problems have been discussed since aspect-oriented programming was introduced. First one is the expressiveness of aspect-oriented programming. To separate more concerns into modules, many studies introduced more expressiveness into pointcuts. Second one is the *fragile pointcut problem* [51]. Aspect-oriented programming helps programmers to change behavior of existing programs from outside the base code. Programmers should specify the structure of the base code to change its behavior. If the base code is edited and its structure is changed, aspect may fail to select the base code. Third one is interference or conflict among extensions. If multiple extensions are applied one place, they may be in conflict. This section describes these three problems and existing proposals.

2.3.1 Expressiveness

To separate more kinds of concerns into modules, expressiveness of destructive extensions should be improved. Several features are proposed to improve expressiveness of aspect-oriented programming languages. For example, new join points and pointcuts for AspectJ are proposed.

Declarative Event Patterns and Tracematch *Declarative event patterns* [57] and *Tracematches* [8] provide history-based pointcuts. With these pointcuts, an advice can be invoked when the given pattern matches on a dynamic execution history. In tracematch, its dynamic execution history is called *trace*. *Trace* consists of entrance and exit of method calls. Figure 2.10 is an example of tracematch. To use tracematch, you define symbols that represent pointcuts, and order them. This tracematch matches the exit of b() if the a() is entered before the exit of b().

History-based pointcuts cannot be used to invoke an advice before a code sequence matching the given pattern starts running, since these pointcuts use execution history. It can be used only to invoke after the code sequence finishes running. They cannot weave around advice into code sequences. Thus, those pointcuts are not appropriate for separating a synchronization

```
1
2  tracematch ( ) {
3     sym a before: call(*.a(..));
4     sym b after: call(*.b(..));
5
6     a b
7
8     {
9        System.out.println("a;b;" );
10    }
11 }
```

Figure 2.10. Example of Tracematch

concern. Pointcuts should know future (like *GAMMA* [36]) to weave around advice into regions.

Loop join points *LoopsAJ* [26] provides a pointcut for selecting a loop join point, which corresponds to a loop body. It allows parallelizing the execution of the specified loop body. Figure 2.11 is a example of LoopsAJ. The former pointcut matches any loop bodies in the whole program. The latter matches *for(int i=MIN; i¡MAX; i+=STRIDE)* style loop bodies; *MIN*, *MAX* and *STRIDE* are passed to arguments of the advice. Since proceed() can be called in another thread, you can parallelize each loop body.

Loop join point is not so powerful to choose loop join points. If a method body contains multiple loops, LoopsAJ cannot distinguish these loops. The loop() pointcuts can distinguish loop join points by args.

A loop join point has another problem. A loop join point may not be applied around advice. If a loop has several successor nodes, it has labeled break, for example (Figure 2.12), you cannot weave around advice.

Synchronized Block Join Points *Synchronized block join points* [61, 62] are proposed to enable selecting synchronized statements as join points in AspectJ. With their work, programmers can select existing synchronized statements. Figure 2.13 shows an example of synchronized block join points. synchronize() pointcut can select synchronized statement. A new construct rm_proceed() calls the block of synchronized statement without synchronization operations. So, you can cancel the synchronized() statement and reimplement another strategy of synchronization. For example, you can use a

```
1
2  void around(): loop(){
3          synchronize(lockObj){
4                  proceed();
5          }
6  }
7
8  void around(int min , int max , int stride):
9      loop() && args(min, max, stride)
10 {
11         Runnable r = new Runnable () {
12                 public void run () {
13                         proceed(min, max, stride);
14                 }
15         } ;
16         new Thread(r).start();
17 }
```

Figure 2.11. Example of LoopsAJ

java.util.concurrent.locks.ReentrantLock object instead of synchronized() statement for locking. However they cannot change the granularity of synchronization nor insert new synchronization code within a method body without a synchronized statement.

EJFlow EJFlow [16] is proposed to improve support of exception handling in AOP. It provides a construct to pick out the flow of an exception. EJFlow introduces a concept named *explicit exception channel* or *EEC*, which ab-

```
1 outer:
2 while(...){
3   while(...){
4     if(cond()){
5       break outer;//exit 1
6     }
7   }//exit 2
8 }
```

Figure 2.12. Example of loop with several successor nodes

```
1
2  void around(Map m): synchronize()
3    &&withincode(*.foo()) && args(m)
4  {
5    try{
6          lock.lock();
7          rm_proceed();
8    }finally{
9          lock.unlock();
10   }
11
12 }
```

Figure 2.13. Example of Synchronized Block Join Point

stracts exception handling. Explicit exception channel is represented as a 5-tuple: exception types, raising sites, handling sites, intermediate sites and exception interface. Exception type is a runtime type of exception to be treated. Raising sites are the locations where the exception specified by an exception type is raised. Handling sites are the locations where the raised exception is handled. Intermediate sites are the methods that call the methods of raising sites but do not handle the exception. Exception interface is the list of exceptions that the EEC signals, it is similar to the throws clause in Java.

EJFlow selects exceptions using explicit exception channel and apply advice. Figure 2.14 shows an example of EJFlow. The exception handling advice is invoked when an exception of Exception1 is raised in method A(), A() is not called from S() and the exception reached call site F().

2.3.1.1 Separation of synchronization concerns in AOP

Aspect-oriented programming can modularize various concerns such as logging and Observer pattern [34] by aspects. As mentioned in Section 2.1.2, synchronization concern is also a good candidate for modularizing as an aspect.

However, AOP languages are not suitable for modularizing synchronization concerns. Granularities of the join points in AspectJ are not suitable for synchronization. In AspectJ, call pointcut selects method invocations or new expressions. When call matches method invocations, it corresponds to

```
1
2  pointcut raisingSite: withincode(public void A());
3  pointcut intermedeateSite : !withincode(public void S());
4  pointcut EEC() : echannel(Exception1, raisingSite, intermedeateSite);
5
6  void ehandler() boundto(EEC1()) catching(Exception1 e):
7  withincode(public void F()){
8  {
9          //exception handling
10 }
```

Figure 2.14. Example of EJFlow

invokevirtual, *invokespecial*, *invokestatic* or *invokeinterface* in JVM's bytecode. If call matches new, it corresponds to a sequence of *new* and *invokespecial*. get and set pointcuts select field accesses. They correspond to *getfield*, *putfield*, *getstatic*, and/or *putstatic*. A join point selected by call, get, or set pointcut corresponds to a single bytecode instruction (method invocations or field accesses). It is too fine-grained for synchronization. A join point selected by execution pointcut is a whole method body and hence it is too coarse-grained. Single get or set operation does not need to be synchronized because they are not critical section. If we synchronize a method call or a method body, it is too coarse to improve concurrency. To implement a synchronization concern, it should be possible to select an arbitrary code region as a join point. For example, programmers should be able to select a code region from some statement to another statement in a method body as they insert a synchronized statement there (like Figure 2.15).

Some programmers might think AspectJ has sufficient expressiveness for implementing a synchronization concern. Programmers can modify a method body and extract a new method for such a code region. Then they can write an around advice with execution pointcut, which runs when that new method is attempted to be invoked. This around advice can execute the new method by proceed within a synchronized statement. However, our goal is to provide several aspects each of which implements a different synchronization policy: fine-grained granularity, coarse-grained, synchronized statement, ReentrantLock object in java.util.concurrent.locks package and so on. Each aspect needs to put a different code region of the method body into a synchronization statement. It is not practical to modify an original method body to extract several new methods for those code regions. Strongly associated

```
1  public void foo(){
2          //do something
3
4          beginCriticalWork();
5          //do critical something.
6          endCriticalWork();
7
8          //do something
9  }
10
11 aspect Synchronize{
12   void around() :
13   <from beginCriticalWork() to endCriticalWork() in method foo>
14   {
15      synchronize(someLockObject){
16        proceed();
17      }
18   }
19 }
```

Figure 2.15. Pseudo code of pointcut to synchronize a code

codes/methods may scatter. The readability of the resulting method body might be decreased. If two code regions intersect with each other, extracting a sub method for each code region is not possible.

2.3.1.2 Separation of exception handling concerns

Exception handling and resource handling are also concerns that are difficult to separate into aspects. try-catch blocks are placed in arbitrary locations within a method. Behavior of regions should be changed in order to exception handling codes. Figure 2.16 show an example of a region which may throw exceptions. The region from new FileReader() to br.close may throws an IOException. If you want to insert exception handling in Java, you should directly insert try-catch clause. AspectJ does not provide means to add exception handling code to such a region either. These concerns are also examples for improving expressiveness of AOP.

```
1  public void readLines(String fileName) throws IOException{
2      //do something...
3      doSomething0();
4
5      List<String> lines=new ArrayList<String>();
6
7      BufferedReader br = new BufferedReader(new FileReader(
           fileName));
8      String line;
9      while((line=br.readLine()) != null){
10         lines.add(line);
11     }
12     br.close();
13
14     //do something...
15     doSomething1();
16 }
```

Figure 2.16. A region that includes file operations may throw exceptions

2.3.2 Fragile Pointcut Problem

Pointcut/advice mechanism in AOP languages causes *fragile pointcut problem* [51]. Pointcuts select join points by specifying names of elements in the base languages. This makes aspects and base programs tightly-coupled. Aspects may work incorrectly if someone refactored base programs. Fragile pointcut problem contains two types of problems:

- Pointcuts fail to select intended join points. This kind of problem is caused by non-local modifications. Necessary concerns implemented in aspects are not applied.

- Pointcuts select unintended join points. Unnecessary behavior modifications are introduced to base programs.

There are typical non-local modifications of programs that break pointcuts: renaming, moving method/class, addition/removal members and changing signatures. If a name of a method or class is renamed, pointcuts including `execution`, `call`, `get` and `set` may fail to pick out intended join points. You can avoid the effect by using wildcards in pointcuts, but it does not essentially solve the issues. Moving methods or classes affect pointcuts using `within` or `withincode`. Existing pointcuts may match added methods,

```
1 module EditorApp {
2   class app.editor ..*;
3   expose to app.editor..* : call(* *(..));
4 }
```

Figure 2.17. Example of *open modules*

but it may not be anticipated. Pointcuts lose the join points by removal of elements, even if the advice with the pointcuts must be executed in the program.

Local modifications may also introduce fragile pointcut problem. If a programmer adds code into existing program, aspects may be applied into the additional code. It may introduce unintended behavior. The writer of aspects can avoid this kind of problem by using within or withincode pointcuts.

Several mechanisms are proposed to make pointcuts robuster. *Open modules* [7, 43] and *XPI* [25] are approaches to specify join points via explicit interfaces of join points. Open modules restrict visibility of join points. To select join points in a class, developer of the class should explicitly expose intended join points to an aspect. Other join points are hidden in the modules.

Figure 2.17 shows an example of open modules for AspectJ. It defines EditorApp module that implements a text editor application. The class clause declares that classes defined under the app.editor package are the members of the module. expose to clause *exposes* declares that call join points in the member classes can be selected the aspects defined the app.editor package. The module EditorApp restricts the scope of join points of the member classes.

Test-based pointcut [50] is a pointcut to select join points using unit tests. A test-based pointcut is written like:

```
test(get(* changedDirName))
```

While unit tests are running, execution histories of test cases, which executes join points matching the specified join points (the field access to changedDirName in this case), are recorded. While executing programs, an execution history is compared to the recorded ones when the program reaches a join point. If the execution history matches, the advice will be executed. This approach dose not automatically solve fragile pointcut problem. Change of a base code may break pointcuts if unit tests and the base code are not consistent. However, unit tests are maintained in practice software development.

Test-based pointcuts can work correctly if unit tests are maintained.

2.3.2.1 Fragile Pointcut Problem with synchronization concerns

Section 2.3.1.1 mentioned modularization of synchronization concerns. However, it introduces a problem, which is how to guarantee that at least one synchronization policy is applied. If no synchronization policy is applied, the program is thread-unsafe; this is a bug. However, if a synchronization policy is implemented as an aspect in AspectJ, we cannot confirm that the synchronization aspect is actually woven and synchronization is performed at runtime. In particular, when the base program is modified later, the existing synchronization aspect might be accidentally made not to work any more due to the fragile-pointcut problem [51]. This is a general problem with using an AspectJ's aspect for implementing an alternative feature in feature modeling [31, 32]. In our scenario, a synchronization concern is a feature in the contexts of feature-oriented programming [11]. The synchronization policies are alternative sub-features of that feature, that is, a set of sub-features one of which must be included. Implementing a feature by an aspect is good practice and it is not a new idea. Since an aspect can be attached and detached at a flexible *join point* to a base program without modifying the base program, *i.e.* due to the obliviousness property of AOP [23], it is a useful tool for implementing a feature [9]. This is definitely true for optional features but not for alternative (or mandatory) features.

2.3.3 Conflict among Destructive Extensions

Interference or conflict among destructive extensions has discussed in the field of aspect-oriented programming. Interference among advices in AOP is discussed in [47, 52, 6]. If multiple advices are applied into the same join point, the advices may affect each other and break programs.

Conflict among introduction of new members is also a problem in AOP [33, 28]. Class extensions in object-oriented programming languages may also cause conflicts among extensions. Methods which have the same name are defined in the same class cause an error or unanticipated results. In some languages, for example Ruby or Objective-C, the method defined/loaded later gets available. Earlier one is removed in such a case. A library including class extensions implies a risk that it crashes other libraries calling the methods that those class extensions redefines.

This is a real problem in Ruby. If you load mathn library, all integer division results in a rational number. However, almost all programs in Ruby expect it results in an integer. Except writing small scripts, programmers have to treat this library with special care.

Library developers can avoid method conflicts to a certain degree by introducing a naming convention. If all method names include a unique prefix or suffix, a risk of conflicts is decreased against other libraries. This approach avoids conflicts of added methods but it is not suitable for method redefinition. Moreover, this approach may degrade the usability of a library.

In order to avoid the conflicts, the scope of destructive class extensions should be limited. Some module systems are proposed to provide scope of destructive class extensions. However they does not solve all problems with class extensions.

Selector Namespaces The concept of *selector namespaces* was introduced by Modular Smalltalk [60]. Selector namespaces allow scoped class extensions hence method conflicts can be resolved to a certain degree. However, they do not preserve the local rebinding property. In selector namespaces, you can add new methods to existing classes but cannot redefine existing methods.

Clasboxes *Classboxes* [13] and *Classbox/J* [12] provide a module named a *classbox*. Classboxes allow programmers to write class extensions to modify a library while not affecting other application programs using that library. A classbox can import a class from another classbox and it can redefine a method of the imported class. The redefinition is visible from not only that importing classbox but also the methods of the classes imported by that classbox. Hence a classbox can override the behavior of its imported classbox; this property is called *local rebinding* in the literature. Classboxes are useful to create a customized version of an existing library, for example, to create the Swing library from the AWT library of Java.

Although a classbox hides method definitions from the outside, it exposes them to classboxes importing it. Thus, if a classbox imports classes from other classboxes and those classes contain conflicting method definitions, the problem occurs. Figure 2.18 shows an example of the problem. *CB0* is a classbox that provides Integer class and its div method returning an integer. Classbox *CB1* provides List class with avg method. To calculate an average, *CB1* imports Integer class from *CB0* and redefines div method to return a rational number. Since avg method calls div method internally, it returns an average of elements by a rational number. *CB2* imports List

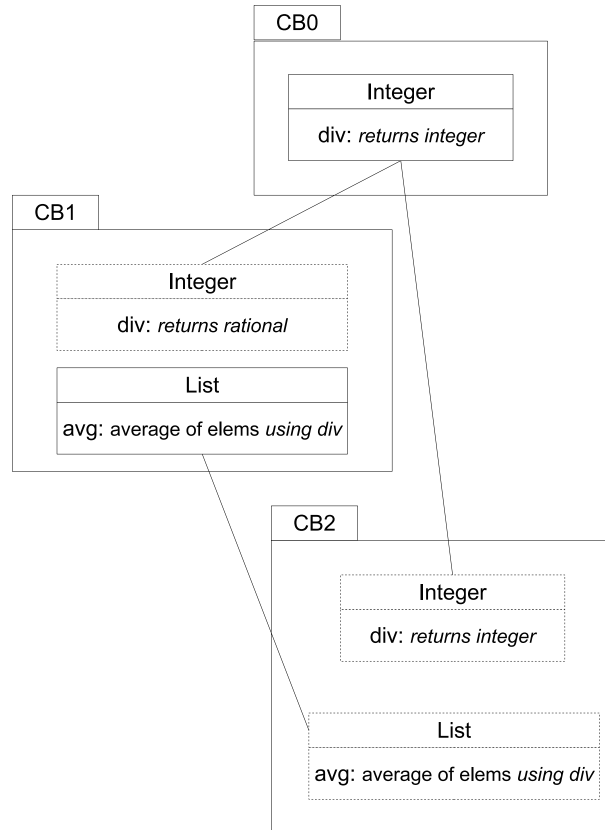


Figure 2.18. A problem in classboxes

from *CB1* to calculate an average, and imports Integer from *CB0* to perform *integer* division. Although the programmer of *CB2* does not know *CB1* internally modifies Integer class, she will expect that avg method returns a rational number. However, Integer from *CB0* overwrites the whole Integer class including the definition of div method due to the local rebinding property. Thus, in *CB2*, avg returns not a rational but an integer number since the two definitions of div conflict in *CB2* and the conflict resolution does not fit the programmer's anticipation.

Refinements Refinement is a module mechanism proposed for Ruby and it will be introduced to ruby 2.0. Refinements confine the scope of class extensions in a module and a user can enable class extensions in a lexical scope. Figure 2.19 shows a sample code using Refinements. A block starting with

```
1  module MathN
2    refine Fixnum do
3      def /(other)
4        Rational(self,other)
5      end
6    end
7  end
8
9  class Foo
10   using MathN
11   def foo()
12     1 / 2
13   end
14 end
15
16 f = Foo.new
17 p f.foo # prints "(1/2)"
18 p(1 / 2) # prints "0"
```

Figure 2.19. Example of Ruby's *Refinements*

refine (line 2) defines class extensions. In the example, the / operator for Fixnum is redefined to return a rational number. Then, the using declaration makes class extensions effective within the current lexical scope. For example, using at line 10 makes the class extensions in MathN effective and hence a call to the / operator at line 12 executes the definition in MathN instead of one in the standard library. However, the class extension in MathN is effective only within the lexical scope from line 9 to 14. If foo method at line 11 calls another method out of this scope and it calls the / operator, then the definition in MathN is not executed. Refinements do not preserve the local rebinding property.

2.3.3.1 Scopes of class extensions

Several scopes can be supposed to restrict scopes of class extensions. This section discusses what a scope is appropriate for class extensions.

Global Global scope is most naive scope for class extensions. Class extensions used in practice have this type of scopes. A benefit of global scope is that it is easy to use class extensions. You do not have to create some special modules

to modify existing programs. A disadvantage is that extensions tend to cause conflicts and there are no means to avoid conflicts. In dynamic languages, it is difficult to find conflicts. With compilers or static checker, you may find conflicts. However, you cannot use multiple libraries, which are in conflict, in the one system.

Lexical Lexical scope for class extensions is used in Ruby's Refinements. With lexical scope, you can use extensions by manually enabling. It is useful to use extensions provided by a library. However, it is not suitable to customize existing libraries. Even if want to change the behavior of an existing library and you redefine a method of the library, the extension is not available in the library's scope. The code of the library cannot call the redefined method. Thus, you cannot change the behavior of the library.

Lexical scope makes class extensions safer and decreases a risk of conflicts among class extensions. However, it restricts expressiveness of class extensions.

Scope with local rebinding property Scope with local rebinding property is proposed by classboxes. This scope can protect methods from other extensions to a certain degree. It also has means to customize existing libraries by class extensions. However, this scope still have a risk of conflict among class extensions by importing multiple modules as mentioned above.

This risk is introduced because the module has only one type of scope. If a module has multiple types of scope and you can use types of scope according to situation. You can protect methods from extensions and you can also customize existing libraries.

2.4 Techniques Similar to Destructive Extensions

2.4.1 Java class loader

Java class loader [39] is a mechanism to load classes dynamically in Java language. Programmers can create custom class loaders and it can be used as name spece of classes. Since every Java class loader makes a separate name space, a different class loader can load a differently declared class with the same name. Although the loaded classes have the same class name, they are treated as the different types at runtime. Created instances are not compatible in the different class loaders. Since a class loaded in a class

```
1  namespace Extensions
2  {
3      public static class StringExtensions
4      {
5          public static int WordCount(this String str)
6          {
7              //...
8          }
9      }
10 }
```

Figure 2.20. Example of a definition of an extension method

loader does not affect classes in other class loaders, Java class loaders do not support class extensions.

2.4.2 C#

C# [21] is a modern object-oriented programming language introduced by Microsoft. C# provides techniques to extend classes: *partial classes* and *extension methods*. Partial classes allow programmers to separate a definition of a class into multiple files. If you add **partial** keyword to class definition, the class's definition can appear in different files. You can separate a concern by partial classes, but you cannot change or add behavior of existing classes.

An extension method is a technique to extend existing classes. An extension method is a static method, which can be called like an instance method. WordCount method in Figure 2.20 is a definition of an extension method. The first argument str is annotated with **this** keyword. It is treated as a receiver. An example of an extension method call is shown in Figure 2.21. To use extension methods, you should import a namespace by **using** directive and you can call an extension method as an instance method. The call of WordCount takes no arguments, but its receiver is passed as the str argument.

Extension methods can extend any existing classes including system-provided classes. However they are not destructive extensions. You can only add new methods and existing methods cannot be replaced. Since extension methods can be enabled only in lexical scope, you cannot change behavior of method calls in classes or files that you can not edit.

```
1  namespace TestApp
2  {
3      using Extensions;
4      class App
5      {
6          static void Main(string[] args)
7          {
8              string s="Lorem ipsum dolor sit amet";
9              System.Console.WriteLine(s.WordCount());
10         }
11     }
12 }
```

Figure 2.21. Example of an extension method call

2.4.3 Virtual classes

BETA [38, 40] and *Newspeak* [14] programming languages provide a mechanism of *virtual classes*. Reference to a class is not statically defined in these languages. You can extend the superclass's nested classes. Figure 2.22 is an example of virtual classes written in Newspeak. A GUI library GUILibrary is defined, and it has Button class as its nested class. ExtendGUI has GUILibrary's subclass ModernGUILibrary as a nested class. ModernGUILibrary extends Button class, which is the superclass's nested class and, provide mordern look and feel.

Virtual classes enable to customize existing libraries. However, they cannot change behavior of objects created in other libraries. Virtual classes do not support other usage of class extensions

2.4.4 Context-oriented programming

Context-oriented programming [30] (COP) is a new technique to behavior of programs that depends on contexts. Context-oriented programming languages allow multiple definitions of a method for the same class.

COP languages introduce a module named *layer*. Additional methods depending on a context is packaged in a layer. You can manually *activate* a layer. While a layer is activated, method definition in the current layer is called instead of the original definition.

Figure 2.23 shows an example of ContextJ, which is a COP language based on Java. layer block defines a layer. You can define the method in the

```
1  class GUILibrary usingplatform: platform = (...)
2  (
3     public class Button = (...)(...)
4     ...
5  )
6
7  class ExtendGUI withGUILibrary: gui(
8     |GUILibrary = gui.|
9  )(
10    public ModernGUILibrary usingPlatform: platform =
11       GUILibrary usingPlatform: platform (
12    )(
13       public class Button = (...)(...) (* a button with modern look and
             feel *)
14       ...
15    )
16 )
```

Figure 2.22. Example of virtual classes in Newspeak

layer block and it can have the same signature of the existing methods. You can use with clause to activate the layer like following code:

```
1     with(MemoryWarning){
2        Image.createImage(name);
3     }
```

The call of createImage method in the block executes the method defined in the MemoryWarning layer. The extent of activation of layers is dynamic. The layer is also activated in the indirectly called methods from the with block.

2.4.5 Implicit conversion in Scala

Scala [42] is a object-oriented and functional programming language. An object in Scala can be implicitly converted into other data type by *implicit conversion*.

Figure 2.24 is an example of implicit conversion in Scala. In main method, variable a refers an Array of Int object and average method is called. Since Array class does not have average method in Scala, it seems to call an undefined method. In such a case, Scala tries to convert other type. intArrayToMyArray function is a function for implicit conversion. It has **implicit** modifier and its

```
1  class Image
2    public static Image createImage(String filename){
3          ...
4          //load and return an image
5    }
6
7    layer MemoryWarning {
8      public static Image createImage(String filename){
9          ...
10         return cachedEmptyImage();
11     }
12   }
13 }
```

Figure 2.23. Example of context-oriented programming in ContextJ

type of the argument is Array[Int] and the return type is MyArray. MyArray class has average method. Scala converts Array[Int] object to MyArray object using intArrayToMyArray function.

You can call average method as if Array class is extended by implicit conversion. However you cannot replace existing methods in Scala.

2.4.6 Type classes

Type classes [56] in functional programming languages also provide a way to add methods or generic functions to existing types. *JavaGI* [58, 59] introduced them into Java.

Example of type classes written in JavaGI is shown in Figure 2.25. SimpleList, EmptyList and Node classes implement a list of strings. Suppose these classes are distributed as a compiled library. You can add methods to these existing classes by using type classes. Count interface declared count method and implementation clauses supply the implementations of the interface. count method can be called in testCount method as if count method is defined directly in SimpleList.

Since new methods can be added into existing class with type classes, they help extension of software. They do not support change of existing methods. In other words, they are not destructive extensions.

```
1  class MyArray(ary:Array[Int]){
2     def average()={
3        (ary.reduceLeft((r,i)=>r+i):Double) / ary.length
4     }
5  }
6
7  object avg{
8     implicit def intArrayToMyArray(ary:Array[Int]):MyArray =new
            MyArray(ary)
9     def main(args:Array[String]):Unit={
10       val a=Array(1,2,3)
11       println(a.average)
12    }
13 }
```

Figure 2.24. Example of implicit conversion in Scala

```
1  abstract class SimpleList {}
2  class EmptyList extends SimpleList {}
3  class Node extends SimpleList {
4      String str;
5      SimpleList next;
6  }
7
8  interface Count {
9      int count();
10 }
11 implementation Count [SimpleList] {
12     int count() {
13         return 0;
14     }
15 }
16 implementation Count [Node] {
17     int count() {
18         return this.next.count()+1;
19     }
20 }
21
22 class Test{
23     void testCount(SimpleList l){
24         System.out.println(l.count());
25     }
26 }
```

Figure 2.25. Example of type classes in JavaGI

	Expressive	Less fragile	Less interference
OOP		✓	✓
Pointcut/advice in AOP	✓		
Class extensions	✓$^{-}$	✓	

Table 2.2. Summary of existing mechanisms.

2.5 Summary

This chapter discussed problems with destructive extensions and motivating examples. These problems — expressiveness, fragile pointcut problem, and interference — should be addressed to replace programs. To support more types of concerns, expressiveness of aspect-oriented programming should be improved. AOP languages should guarantee application of mandatory concerns or get less fragile. Class extensions should have means to control scope of extensions and avoid interferences. However, as shown in Table 2.2, existing mechanisms do not address these problems. OOP is less expressive than AOP and class extensions, however, it is safer than them. AOP is expressive, however, it got fragile and interferences. Class extensions got also expressive, and it introduces a risk of interferences. Of course, AOP and class extensions are more expressive than OOP, however, they should be more expressive to deal with more types of concerns.

In the following chapter, this thesis proposes language constructs to address these problems. Regioncut makes AOP more expressive. An assertion for advice makes AOP less fragile. Method shelters provide means to avoid interferences among class extensions.

Chapter 3

Regioncut

To improve expressiveness of aspect-oriented programming, this chapter proposes a new construct named *regioncut* [5, 3] for AspectJ. A regioncut behaves like pointcuts but it selects code regions, not execution points. It statically determines the selected regions at compile-time. A regioncut helps to separate various concerns such as synchronization, exception handling, and transaction.

3.1 Overview

The syntax of a regioncut is simple. A regioncut takes an ordered list of pointcut designators separated by a comma and then it selects the code region including the join points selected by every pointcut designator in that order. Only the call, get and set pointcut designators are available for a regioncut, which statically select a single expression. A regioncut selects a code region within a method body; it does not select a code region stretching over multiple method bodies.

The code shown in Figure 3.1 is an example of regioncut. The parameter to this regioncut is a list of two pointcut designators. It selects a code region that starts with a method call to List.get(int) and ends with a field access to Foo.bar.

```
1 pointcut rc1():
2   region[
3     call(Object List.get(int)),
4     get(int Foo.bar)
5   ];
```

Figure 3.1. A regioncut designator

```
1 pointcut rc2():
2   region[
3     call(Object List.get(int)),
4     set(* Foo.foo),
5     get(int Foo.bar)
6   ];
```

Figure 3.2. A regioncut designator can have three or more pointcuts.

A parameter to a regioncut can be a list of more than two pointcut designators (in Figure 3.2). Now the regioncut selects a code region in which the List.get(int) method is first called, then the Foo.foo field is set, and finally the Foo.bar field is read. The code region may contain other statements and expressions between the call to the List.get method and the access to the Foo.bar field. The access to the Foo.foo field is not the only expression between them.

Specifying an intermediate join point like set(* Foo.foo) is useful to distinguish similar code regions in the same method. For example, the pointcut rc1 shown above matches two regions in the method in Figure 3.3. If we want to select only the region from line 4 to 6, we must use the pointcut rc2 instead. The region from line 10 to 12 is excluded.

A code region selected by a regioncut is a collection of consecutive statements. The boundary of the code region is never in the middle of a statement. Suppose that a join point selected by an argument to the regioncut, for example, a method call is a term of some long expression. In this case, the selected code region would be expanded to include the whole statement for that long expression. If the selected join point is in the else block of an if statement, as we below describe, the whole if statement may be included in the selected code region.

You can apply before, after and around advice to the selected code region.

```
1  void methodWithSimilarRegions(List l,int i,Foo f){
2    int n;
3
4    Object o=l.get(i);
5    f.foo=o;
6    n= f.bar;
7
8    //do something
9
10   Object o=l.get(i+1);
11   System.out.println(o);
12   n= f.bar;
13 }
```

Figure 3.3. A method including two similar regions

Since code regions do not have return values, the return type of the applied around must be void.

3.2 Semantics

The semantics of regioncuts is shown in Figure3.4 and Figure 3.5 in Scheme-like code. In our implementations, regioncuts support all control structures including do-while and try-catch-finally statements. However, for the sake of ease, we consider only if-else and while statements.

The evaluation of methods is done by the eval-method function. method-body and eval-stmts, which is the arguments of the function, are sequences of the statements. eval-stmts function is the main evaluation function. It takes following-stmts argument, which contains statements to be evaluated next. This is the key characteristic of this semantics.

First, eval-stmsts function in Figure 3.5 looks up a regioncut that matches stmts, the given sequence of statments. If it is found, evaluate the advice corresponding to the found regioncut. If not, the last statement in the stmts will be removed and the function looks up a regioncut again. The removed last statement will be inserted into the head of following-stmts. If length of the stmts become one, evaluate the statement. Then following-stmts become stmts and call eval-stmts again. For example, the sequence of statements are tested from 1 to 7 in Fgure 3.6.

3.2.1 Matching between regioncuts and a sequence of statements

If join points which match a pointcut in the sequence of pointcuts are appeared in the sequence of statements with the same order, the statements match the regioncut. The first pointcut in the regioncut must match the first statement and the last pointcut must match the last statement.

The matching is done from larger region. If the first and last matched statements are in the blocks of different nesting level, the block that includes both blocks is selected as the matched region. Thus a imcomplete region, for example whose head or tail is in the if block, is not selected by regioncuts. For example, the regioncut `region[ call(* *.a()), call(* *.b()), call(* *.d())]` matches the following block:

```
1 {
2     a();
3     if(cond){
4         b();
5     }else{
6         c();
7     }
8     d();
9 }
```

match function in Figure 3.7 is the matching function of this semantics. It tests whether regioncut rc matches stmts, the list of statements, or not. An element of stmts is an expression statement, a while statement, a if statement or a block, which is the nested list of statements.

match* function tests whether one statement stmt matches regioncut rc, which is the list of pointcuts. If stmt is while statement, its conditional expression and body are treated as consecutive statements. Matching is done in the order of the conditional expression to the body. If stmt is if statement, its conditional expression and its then block are treated as consecutive statements and its conditional expression and its else block. The both then and else block with conditional expression is tested. The block which matches more pointcuts is adopted.

```
(define (eval−method method−body env)
          (eval−stmts method−body '() '() env))

(define (eval−stmts stmts following−stmts proceed env)
    (let ((advice (lookup−advice stmts)))
        (cond (advice
                    (eval−advice advice stmts) ; evaluate <advice> if there
                          are matched advice
                    (eval−stmts following−stmts '() proceed env)) ;
                         evaluate following statements
                ((= (length stmts) 1)
                    (eval−statement (car stmts)) ; evaluate the statement
                         if it is one statement
                    (eval−stmts follwoing−stmts '() proceed env)) ;
                         evaluate following statements
                 ((> (length stmts) 1) ; if there are multiple statements
                    ; add last statement to the head of following
                         statements
                    ; and then re−evaluate
                    (eval−stmts (take stmts (− (length stmts) 1))
                                   (cons (last stmts) follwoing−stmts)
                                        proceed env)))))

(define (lookup−advice stmts) ; return advice which matches sequence
     of statements
    (find all−advice−regioncut−pairs
          (lambda (advice regioncut) (match stmts regioncut))))

(define (eval−advice advice proceed env) ; evaluate <advice>
   (eval−stmts (advice−body advice) '() proceed env))
```

Figure 3.4. eval-method function for evaluating a program with reginocut

```
(define (eval−statement stmt proceed env)
  (cond ((expr? stmt) (eval−expr stmt env))
        ((if? stmt)
          (if (eval−expr (cond−part stmt) env) ; evaluate "if"
              statement
            (eval−stmts (then−part stmt) '() proceed env)
            (eval−stmts (else−part stmt) '() proceed env)))
        ((while? stmt)
          (cond ((eval−expr (cond−part stmt) env) ; evaluate "while"
                 statement
                   (eval−stmts (while−body stmt) '() proceed env)
                   (eval−statement stmt proceed env))))
        ((block? stmt)
          (eval−stmts (block−body stmt) '() proceed env)) ; evaluate
              block
        ((proceed? stmt)
          (eval−stmts proceed '() '() env))))
                          ; evaluate "proceed()" in advice
```

Figure 3.5. eval-stmt function for evaluating a statement with reginocut

```
void foo(){
    a();
    b();
    If(...){
        c();
    }
}
```


Figure 3.6. Range and order of evaluation with reginocut

```
;whether stmts matched rc or not
(define (match stmts rc)
  (let ((rc−rest (match∗ (car stmts) rc)))
    (if (eq? rc−rest rc) ; whether the head stmts does not the head of
        rc
        #f   ; false if not
        (null? (match−seq (cdr stmts) rc−rest))))) ;true if rc matches
            until the tail of stmts

;tests whether stmts matches rc (seq of pointcut)
;returns seq of remaining pointcuts that does not match
(define (match−seq stmts rc)
  (if (null? stmts)
    rc ; do matching from the head
    (match−seq (cdr stmts) (match∗ (car stmts) rc))))

;do matching between stmt and returns remaining pointcuts that does
    not match
(define (match∗ stmt rc)
  (cond ((expr? stmt) (if (include−jp? stmt) (cdr rc) rc))
        ((while? stmt) ; case of while, do matching from conditional to
              body
          (let ((rc−after−cond (match∗ (cond−part stmt) rc)))
                        (match∗ (body−part stmt) rc−after−cond))
        ((if? stmt)  ; case of if, do matching conditional and both do
            then and else block
                (let ((pc−rest (match∗ (cond−part stmt) rc)))
             (shorter (match∗ (then−part stmt) rc−rest)
             ; shorter remaining pointcuts are returned
                      (match∗ (else−part stmt) rc−rest))))
        ((block? stmt) (match−seq (block−body stmt) rc))))
```

Figure 3.7. Matching function match for regioncut

3.3 Context Exposure

In Java, a synchronized statement takes an object that will be locked. To implement a synchronization concern by an aspect, the object must be available within an advice body.

A regioncut can be used with other pointcut designators including this, args, and target. If a locked object is stored in a field of this object, the this pointcut can be used to obtain this object. If a locked object is in an argument or a target object, the args or target pointcut can be used to obtain it. If the value bound to the parameter to args or target is from a local variable or a field and the variable (or a field) is available at the beginning of the code region, the value of the variable (or a field) at the beginning of that region is passed to an advice body as an argument. Otherwise, if the variable is not available at that point, or if the argument to args or target is a compound expression, then a compile error is reported.

Figure 3.8 is an example of context exposure. The regioncut for the around advice selects a code region from line 5 to 7. The argument to b at line 6 is taken from a local variable i, which is initialized before the call to a at line 5, and the target object of the field access at line 7 is directly taken from the obj field. Hence the advice parameters o and n are bound to the values of i and obj at line 5.

3.4 Implementation Issues

We implemented regioncut for AspectJ as an extension to the AspectBench Compiler (abc) [10]. The intermediate language of abc is Jimple [55]. We use Jimple for pattern matching for regioncut.

3.4.1 Analysis of blocks and statements

Selected regions are extpanded to fit control structures as mentioned in Section 3.2. Since Jimple has no information about where blocks, statements, and control structures start and end, exact regions of control structures can not be recognized. We extended Jimple to make this information available.

This thesis introduced a new Jimple instruction *marker*. A marker has two properties. One is a kind of the structure, which is either statement, block, if, while, or others. The other is whether the marker represents beginning or ending. We modified the Java-to-Jimple compiler so that a pair

```
1  class Foo{
2    SomeObject obj;
3    void bar(){
4      int i=10;
5      a();
6      b(i);
7      obj.c;
8    }
9  }
10
11 void around(SomeObject o, int n):
12   region[
13     call(* *.a()),
14     call(* *.b(int)) && args(n),
15     get(* SomeObject.c) && target(o)
16   ]
17 {
18   proceed(o,n);
19 }
```

Figure 3.8. Context exposure by a regioncut

of marker will surround all the instructions of each statement. Other structures such as a block are also surrounded by a pair of markers. Java code is internally compiled to Jimple like Figure 3.9.

After the region expansion, all the inserted markers are removed and then Jimple instructions are converted into Java bytecode.

3.4.2 Around advice support

To implement proceed in an around advice, the abc compiler extracts a new static method from the code corresponding to a join point shadow [41], for example, one selected by execution and initialization pointcuts. The values of method parameters and so on at the join point shadow are passed as arguments to the static method. We extend this implementation technique for supporting an around advice with a regioncut.

Assignments to local variables Suppose that a static method is extracted for a code region selected by a regioncut. If a new value is assigned to a local variable within that code region and that variable is declared out of the

region, then the new value must be first stored in another local variable in the extracted static method and then reflected on the original local variable.

To implement this behavior, we make an object whose fields are copies of the local variables accessed in the code region. It is passed to the extracted static method and, if some fields of the object are updated in the static method, then the updated values are copied back to the original local variables.

For example, in Figure 3.10, the code region between the two method calls to a and b is selected and hence a static method is extracted from that region. The assignment to a local variable s at line 4 must be transformed so that the value assigned at line 4 is reflected on the value of s at line 6. Before the aspect is woven by the original weaver of abc, therefore, the toBeAdvised method is transformed into the code shown in Figure 3.11. The class LocalStorage$toBeAdvised is a helper class generated during this transformation. The static method extracted for the region from line 6 to 12 receives the value of *$localStorage* as an argument.

A helper class such as LocalStorage$toBeAdvised is generated per code region. Each field has the same type as the corresponding local variable. We do not use a java.util.HashMap object or an Object array for *$localStorage*. These are more generic but type conversion or boxing/unboxing is needed when a value is stored and obtained from them.

Jumps to the outside of the region Jumps — break, continue, and return statements — must be also transformed when a static method is extracted from a code region selected by a regioncut. The destination of these jumps may be out of that region. Figure 3.12 shows a program including jumps to out of the selected region. Note that, in Jimple and Java bytecode, break and continue statements are represented by goto instructions.

For the transformation, each jump instruction from the inside to the outside of the region is given a unique identification (id.) number. Next, each jump instruction is replaced with the following instructions:

1. Save the id. number into a local variable (jump id. variable).

2. Jump to the end of the region.

Furthermore, at the end of the region, a switch statement is inserted. It branches to the destination specified by the jump id. variable. Figure 3.13 shows the resulting program after the transformation above. Then our extended abc compiler extracts a static method from the code region from line

5 to 16 in Figure 3.13. The abc compiler is responsible to maintain the consistency of the value of the jump id. variable __i between the extracted static method and the original method includeJump. A return statement is transformed in a similar way.

Java code

```
1  public static int fact(int n){
2     if(n<=0){
3        return 1;
4     }else{
5        return n * fact(n−1);
6     }
7  }
```

Pseudo Jimple code

```
1  public static int fact(int n){
2     BLOCK:begin:0;
3        IF:begin:1;
4              STMT:begin:2;
5                 if n > 0 goto BLOCK:begin:5;
6              STMT:end:2;
7           BLOCK:begin:3;
8              STMT:begin:4;
9                 return 1;
10             STMT:end:4;
11          BLOCK:end:3;
12          BLOCK:begin:5;
13             STMT:begin:6;
14                $i0 = n − 1;
15                $i1 = fact($i0);
16                $i2 = n * $i1;
17                return $i2;
18             STMT:end:6;
19          BLOCK:end:5;
20       IF:end:1;
21    BLOCK:end:0;
22 }
```

Figure 3.9. Example of *markers*

```
1  public void toBeAdvised(int x){
2      String s="initial string";
3      a();
4      s="string was replaced";
5      b();
6      System.out.println(s); // what is the value of s?
7  }
8
9  void around():
10     region[
11         call(* *.a()),
12         call(* *.b())
13     ]
14 {
15     proceed();
16 }
```

Figure 3.10. A local variable is updated within a code region

```
1  public void toBeAdvised(int x){
2      String s="initial string";
3      $localStorage = new LocalStorage$toBeAdvised();
4
5      $localStorage.s=s;
6      nop; //label for the beginning of the region
7      s=$localStorage.s;
8      a();
9      s="string was replaced";
10     b();
11     $localStorage.s=s;
12     nop; // label for the end of the region
13     s=$localStorage.s;
14
15     System.out.println(s);
16 }
```

Figure 3.11. A transformed version of the method in Figure 3.10

```
1  public void includeJump(){
2      labelOfFor:
3      for(;;){
4          while(true){
5              a();
6              if(innerBreak()){
7                  break; //goto label0;
8              }else{
9                  break labelOfFor; //goto label1;
10             }
11             b();
12         }
13         //label0:
14     }
15     //label1;
16 }
17
18 void around(): region[call(* *.a()),call(* *.b())] {
19     proceed();
20 }
```

Figure 3.12. A program including jumps to out of a selected code region

```
1  public void includeJump(){
2      labelOfFor:
3      for(;;){
4          while(true){
5              nop; //label for beginning of the shadow
6              a();
7              if(innerBreak()){
8                  __i=0;
9                  goto endLabel;
10             }else{
11                 __i=1;
12                 goto endLabel;
13             }
14             b();
15             endLabel:
16             nop; //label for end of the shadow
17             switch(__i){
18             case 0: goto label0;
19             case 1: goto label1;
20             }
21         }
22         label0:;
23     }
24     label1:;
25 }
```

Figure 3.13. A transformed version of the method in Figure 3.12

3.5 Evaluation

Regioncut is implemented by extending the AspectBench 1.3.0 compiler with the JastAdd frontend [29, 22] running on Sun JVM 1.6. Then we evaluated the design of the regioncut and the assertion for advice by applying them to two open-source software products.

3.5.1 Javassist

We wrote synchronization aspects for Javassist by using the proposed constructs along the scenario in Section 1.1. One aspect implements fine-grained synchronization. Figure 3.14 shows an equivalent program in which fine-grained synchronization code is embedded by hand. Two synchronized statements are embedded in the reateClass2 method. Figure 3.15 shows an equivalent program written by hand for coarse-grained synchronization. One ynchronized statement is embedded in the createClass method, which calls createClass2. We could successfully separate the synchronization code into aspects by using regioncut. Figure 3.16 and Figure 3.17 show (the advices in) the aspects.

The Javassist users can easily switch synchronization policies by selecting either of the two aspects. Switching the policies caused performance differences according to our experiment. We ran the benchmark test posted with the bug report [1]. It is a client-server program, in which Javassist is used for the client-side code running 20 threads. For our experiment, we used machines with Intel Xeon (2.83 GHz), Linux 2.6.28 (x64), and Sun JVM 1.6.0. The client machine had 8GB memory and the server one had 4GB memory. They are connected through 1Gbps Ethernet. We disabled assertion for advice while running this benchmark.

Table 3.1 lists the results. We used two kinds of client machine: one with 4 cores and the other with 2 cores. The numbers are the average of the execution time measured 500 times. The results revealed that using a fine-grained synchronization aspect is better on the 4 core machine while using a coarse-grained one is better on the 2 core machine. The overheads due to using an aspect were negligible.

3.5.2 Hadoop

Hadoop [54] is an open-source framework for distributed computing; it provides a distributed file system and programming supports for the MapReduce

	Time (sec.)	Std. Deviation
—quad core		
fine-grain (by aspect)	5.70	0.13
fine-grain (by hand)	5.63	0.13
coarse-grain (by aspect)	7.77	0.26
coarse-grain (by hand)	7.87	0.33
—dual core		
fine-grain (by aspect)	9.94	0.21
fine-grain (by hand)	9.94	0.21
coarse-grain (by aspect)	8.70	0.20
coarse-grain (by hand)	8.76	0.24

Table 3.1. The execution time of the Javassist benchmark

computing model [20]. We rewrote the program of Hadoop 0.16.4 in AspectJ with our proposed constructs.

We separated synchronization concerns into aspects from the TaskTracker class (2357 LOC) in org.apache.hadoop.mapred package of Hadoop. Table 3.2 lists the result of our experiment. The TaskTracker class contains 21 synchronized statements. We separated all the statements into aspects. Among them, 9 statements could be separated into aspects by using ordinary pointcut designators such as call, get, set, or execution. We needed the regioncut to separate the rest of the synchronized statements into aspects. Figure 3.18 and Figure 3.19 show the example methods and advice which need regioncut. Note that we did not modify the original source program of the TaskTracker class. If we performed refactoring to extract a new method from the synchronized block, then we would need less regioncuts for separating synchronized statements into aspects.

We also evaluated the necessity of more than two arguments to a regioncut. Recall that a regioncut can take more than two pointcuts as arguments to distinguish similar code regions in the same method body. Among 12 synchronization concerns in the TaskTracker class, 5 concerns needed regioncuts that take more than two pointcuts as arguments. Furthermore, 4 concerns needed our proposed context exposure mechanism.

synchronized statements	21
ones separated by ordinary pointcuts	9
ones separated by regioncuts	12

Table 3.2. The number of synchronization concerns in the TaskTracker class

3.6 Current Limitation

Two regioncuts may select two code regions intersecting each other. Our compiler cannot implement around advices for those two regions. In Figure 3.20 the region from beginA() to endA() and the region from beginB() to endB() are intersection. We cannot apply around advice these regions. The compiler can implement if one of the selected regions is nestedly contained in the other region. We should define the priority rule of intersected regions to implement around advices for such regions. For example, if we define a rule — a precedent region should be enlarged to contain following regions —, we can solve this problem.

```
1  private static WeakHashMap proxyCache;
2  private void createClass2(ClassLoader cl) {
3    CacheKey key = new CacheKey(superClass, interfaces, methodFilter,
         handler);
4    synchronized (proxyCache) {
5      HashMap cacheForTheLoader = (HashMap)proxyCache.get(cl);
6      if (cacheForTheLoader == null) {
7        cacheForTheLoader = new HashMap();
8        proxyCache.put(cl, cacheForTheLoader);
9        cacheForTheLoader.put(key, key);
10     } else {
11       CacheKey found = (CacheKey)cacheForTheLoader.get(key);
12       if (found == null)
13         cacheForTheLoader.put(key, key);
14       else {
15         key = found;
16         Class c = isValidEntry(key); // no need to synchronize
17         if (c != null) {
18           thisClass = c;
19           return;
20         }
21       }
22     }
23   }
24
25   synchronized (key) {
26     Class c = isValidEntry(key);
27     if (c == null) {
28       createClass3(cl);
29       key.proxyClass = new WeakReference(thisClass);
30     }
31     else
32       thisClass = c;
33   }
34 }
```

Figure 3.14. Fine-grained Synchronization by hand

```
1  public Class createClass() {
2     if (thisClass == null) {
3        ClassLoader cl = getClassLoader();
4           synchronized (proxyCache) {
5              if (useCache)
6                 createClass2(cl);
7              else
8                 createClass3(cl);
9           }
10    }
11
12    return thisClass;
13 }
```

Figure 3.15. Coarse-grained Synchronization by hand

```
1  void around():
2     region[
3        call(* WeakHashMap.get(..)),
4        call(* WeakHashMap.put(..))
5     ]
6  {
7     synchronized(ProxyFactory.class){
8        proceed();
9     }
10 }
11
12 void around(Object key):
13    region[
14       call(* *.isValidEntry(*)) && args(key),
15       set(* *.proxyClass)
16    ]
17 {
18    synchronized(key){
19       proceed(key);
20    }
21 }
```

Figure 3.16. The advices for fine-grained synchronization

```
1  void around():
2     region[
3        get(static boolean *.useCache),
4        call(* *.createClass2(..))
5     ]
6  {
7     synchronized(ProxyFactory.class){
8        proceed();
9     }
10 }
```

Figure 3.17. The advice for coarse-grained synchronization

```
1   private synchronized void purgeJob(KillJobAction action) throws
        IOException {
2     String jobId = action.getJobId();
3     LOG.info("Received 'KillJobAction' for job: " + jobId);
4     RunningJob rjob = null;
5     synchronized (runningJobs) {
6       rjob = runningJobs.get(jobId);
7     }
8
9     if (rjob == null) {
10      LOG.warn("Unknown job " + jobId + " being deleted.");
11    } else {
12      synchronized (rjob) {
13        for (TaskInProgress tip : rjob.tasks) {
14          tip.jobHasFinished(false);
15        }
16        if (!rjob.keepJobFiles){
17          fConf.deleteLocalFiles(SUBDIR + Path.SEPARATOR +
                JOBCACHE +
18                                      Path.SEPARATOR + rjob.getJobId
                                            ());
19        }
20        rjob.tasks.clear();
21      }
22    }
23
24    synchronized(runningJobs) {
25      runningJobs.remove(jobId);
26    }
27  }
28
29  //synchronization advice for the formaer synchronized statement
30  //in purgeJob()
31  void around(Object syncObj):region[
32    get(Set *.tasks) && target(syncObj),
33    call(* JobConf.deleteLocalFiles(String)),
34    call(* java.util.Set.clear())
35
36  ] && withincode(* TaskTracker.purgeJob(KillJobAction))
37  {
38    synchronized(syncObj){
39      proceed(syncObj);
40    }
41  }
```

Figure 3.18. Example Code from Hadoop and Its Synchronization Aspect 1

```
1   private void killOverflowingTasks() throws IOException {
2     long localMinSpaceKill;
3     synchronized(this){
4       localMinSpaceKill = minSpaceKill;
5     }
6     if (!enoughFreeSpace(localMinSpaceKill)) {
7       acceptNewTasks=false;
8       synchronized (this) {
9         TaskInProgress killMe = findTaskToKill();
10
11        if (killMe!=null) {
12          String msg = "Tasktracker running out of space." +
13            " Killing task.";
14          LOG.info(killMe.getTask().getTaskId() + ": " + msg);
15          killMe.reportDiagnosticInfo(msg);
16          purgeTask(killMe, false);
17        }
18      }
19    }
20  }
21
22   void around(TaskTracker t):region[
23    call(TaskInProgress TaskTracker.findTaskToKill()),
24    call(void TaskTracker.purgeTask(..))
25  ] && withincode(* TaskTracker.killOverflowingTasks()) && this(t)
26  {
27    synchronized(t){
28      proceed(t);
29    }
30  }
```

Figure 3.19. Example Code from Hadoop and Its Synchronization Aspect 2

```
1  void withConflict(){
2     beginA();
3     beginB();
4     endA();
5     endB();
6  }
7
8  void around():
9     region[call(∗ ∗.beginA()),call(∗ ∗.endA())]
10 {
11    proceed();
12 }
13
14 void around():
15    region[call(∗ ∗.beginB()),call(∗ ∗.endB())]
16 {
17    proceed();
18 }
```

Figure 3.20. Example of conflict of regioncuts

Before removing:

```
1 {
2     int x = r.getX();
3     int y = r.getY();
4     update(x,y);
5 }
```

After removing:

```
1 {
2     int x = r.getX();
3     update(x,x);
4 }
```

Figure 3.21. A region before/after removing a method call

3.7 Fragility of regioncuts

Regioncuts are fragile when the code is refactored. According to [51], four types of non-local changes may make pointcuts fragile: *rename*, *move method/class*, *add/delete method/field/class*, and *signature changes*. These changes also make regioncuts fragile. If a join point specified in a regioncut is affected changes, the regioncut fails to select its intended region. However, local changes also make regioncuts fragile.

- ***Remove or change a join point:*** If a join point specified in a regioncut is remove or changed, regioncut cannot select an intended region. For example, the former region in Figure 3.21 is before removing a join point. The following regioncut can select the former region:

  ```
  1 region[call(int *.getX()), call(int *.getY()), call(void update
        (..)]
  ```

  The latter region is the region after removing the method call of getY(). In this case, the regioncut does not match the region. Even if the call of getY() is changed to other method, the regioncut also does not match the region. This change is local since the removal or change method calls does not change method's interface. However, this change makes regioncuts fragile.

- ***Reorder join points:*** Reordering of join points also makes regioncuts

Before reordering:

```
1 
2     int x = r.getX();
3     int y = r.getY();
4     update(x,y);
```

After reordering:

```
1 
2     int y = r.getY();
3     int x = r.getX();
4     update(x,x);
```

Figure 3.22. A region before/after reordering a method call

fragile. Figure 3.22 shows an example of regioncut. The following regioncut can select the former region in the Figure:

```
1 region[call(int *.getX()), call(int *.getY()), call(void update
      (..)]
```

The methods getX() and getY() are getter methods; they do not cause side-effect. The meaning of the program does not change if order of the method calls is changed. Although this change is local, the regioncut fails to select the latter region since regioncuts depend on order of join points.

In order to make regioncuts more robust, there is an approach that more flexible mechanisms to specify regions will be introduced. However, it makes regioncuts more complex. We will introduce an assertion for advice in the next chapter, to find regioncuts that fails to select intended regions.

3.8 Summary

This chapter presents a new language construct for aspect-oriented programming named *Regioncut*. Regioncut is means to select a code region instead of a join point. It improves expressiveness of AOP or destructive extensions. You can separate synchronization concerns into an aspect and switch granularity synchronization. Regioncut can also separate exception handling concerns (showed in Section 2.3.1.2). Figure 3.23 shows an aspect that

```
1  aspect ExceptionHandling{
2      void around():
3        region[
4          call(BufferedReader.new(..)),
5          call(void BufferedReader.close())
6        ]
7      {
8          try{
9              proceed();
10         }catch(IOException{ e){
11             //handle an exception
12         }
13     }
14 }
```

Figure 3.23. An advice that handles an exception thrown from Figure 2.16

handles exception by using regioncut. This aspect handles an IOException thrown from a region in Figure3.23.

Transactional pointcuts [49] [1] are pointcuts similar to regioncuts. Transactional pointcuts can also select code regions by specifying a sequence of pointcuts. Motivating examples of transactional pointcuts are exception and resource handling.

This chapter showed semantics of regioncut. Regioncut is applied to Hadoop and Javassist in this chapter. synchronized statements in a class of Hadoop can be separated into an aspect by regioncuts. Regioncut enables to switch synchronization in Javassist and to improve performance.

[1] Transactional pointcuts are proposed at the same conference which regioncut is presented.

Chapter 4

An Assertion for Advice

To address fragile pointcut problem in aspect-oriented programming languages, this chapter proposes a new assertion mechanism for AspectJ named an *assertion for advice* [3]. This enables programmers to test an assumption that a certain advice is woven and it modifies the program behavior at some execution point. This mechanism is useful in particular for an advice with a regioncut, which tends to be fragile. Even small changes of a base program may make the regioncut not to match the code region where the advice must be woven. The proposed mechanism would make programmers less reluctant to use an aspect to implement an alternative feature such as a synchronization concern. This mechanism addresses a certain degree of fragile pointcut problem that pointcuts fail to select intended join points. A problem, which pointcuts may select unintended join points, can be avoided if a programmer uses within or withincode pointcuts. Hence this mechanism does not deal with this type of problem.

4.1 Overview

This chapter proposes two kinds of annotations: @AssertAdvised and @SolveProblem. The former annotates a method and the latter annotates an advice. Figure 4.1 and Figure 4.2 show examples. @AssertAdvised declares

```
1 class A{
2   @AssertAdvised("name_of_problem")
3   void foo(){
4     bar();
5   }
6 }
```

Figure 4.1. An annotation for a method

```
1 @SolveProblem("A.name_of_problem")
2 void around(): call(* *.bar()) {
3   //do something
4 }
```

Figure 4.2. An annotation for an advice

that the behavior of the annotated method, for example, the foo method in Figure 4.1, must be modified for implementing some concern by an advice annotated by @SolveProblem, for example, the advice in Figure 4.2. The argument to @SolveProblem represents which problem in which class the advice is expected to solve. For example, @SolveProblem("A.name_of_problem") solves the problem in the method with @AssertAdvised("name_of_problem") in class A.

@AssertAdvised tests if the annotated method satisfies the following assumption just before the method returns:

- the method is directly or indirectly invoked (through proceed) from the @SolveProblem advice (*i.e.* the method is under the control of the advice, shown in Figure 4.3), or
- the method directly or indirectly invokes the @SolveProblem advice while the method is being executed (*i.e.* the part of the method body is under the control of the advice, shown in Figure 4.4).

Here the @SolveProblem advice is an advice with the @SolveProblem annotation corresponding to the @AssertAdvised of the method. If the test fails, then java.lang.AssertionError will be thrown.

We did not choose simpler design, in which @AssertAdvised tests if a specific advice is woven at a specific join point. The reason is to allow refactoring

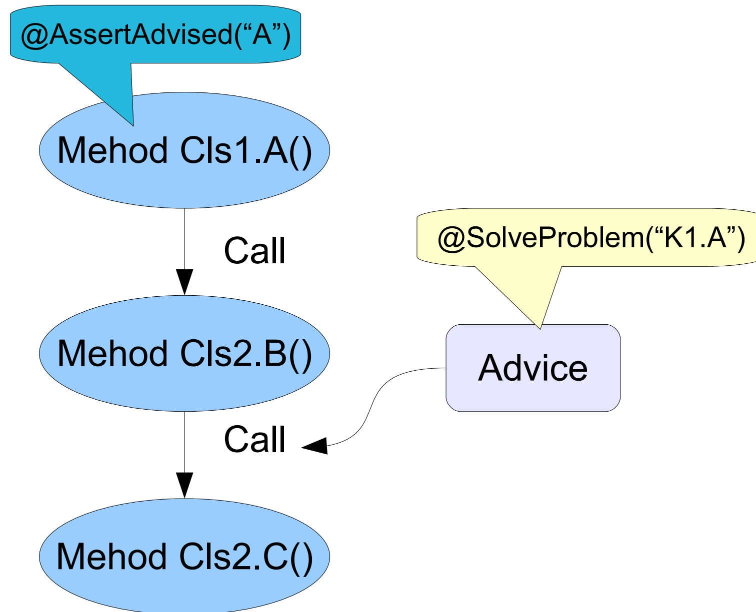


Figure 4.3. Advice with @SolveProblem is invoked by method with @AssertAdvised

on the advice. For example, a synchronization concern can be implemented with different policies. Thus, programmers might replace an original synchronization aspect with a new one they write. The new aspect might be woven at a different join point. @AssertAdvised must consider such a new aspect is woven at a different join point. The design of @AssertAdvised presented in this chapter uses a higher level abstraction and accept such refactoring on aspects.

Note that an @AssertAdvised annotation is not inherited by a subclass. Suppose that a class has a method annotated with @AssertAdvised and its subclass overrides that method. The @AssertAdvised annotation is not added to the overriding method in the subclass unless another @AssertAdvised annotation is explicitly written for the overriding method. This is because the implementation of the overriding method might be different and thus the advice is not needed any more or another kind of advice is needed.

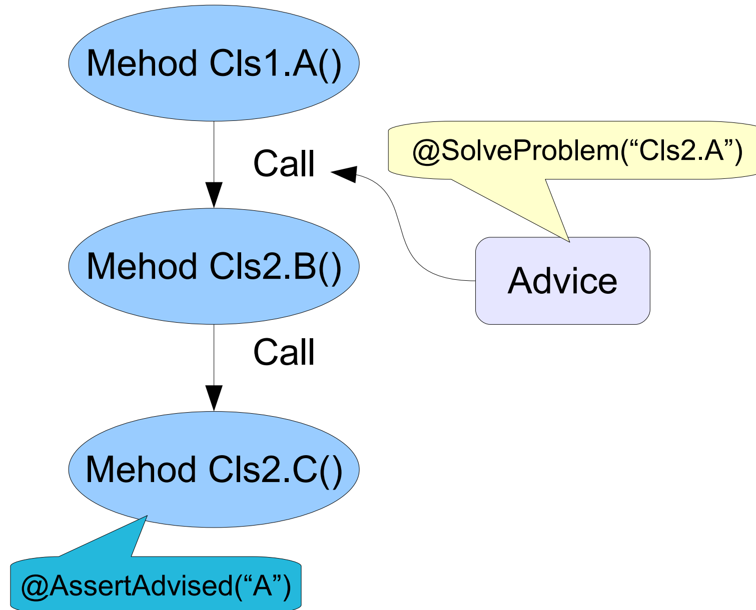


Figure 4.4. Method with @AssertAdvised is invoked by advice with @SolveProblem

4.2 Implementation

We implemented assertion for advice by modifying the compiler implemented for regioncut. The tests are done during compilation. The compiler analyzes Jimple and creates a call graph.

Fisrt the tester loads class files and creates the call graph. Then the tester checks a methods annotated with @SolveProblem has a possibility of calling a method which has @AssertAdvised annotation with the same concern name, or a method with @AssertAdvised has possibility of calling a method with @SolveProblem. Then, the tests pass, otherwise, the compiler shows warning message.

4.3 Evaluation

To evaluate the assertion for advice, we added @AssertAdvised and @SolveProblem annotations for the synchronization aspects separated by regioncuts from the TaskTracker class shown in Section 3.5.2. We then updated the program from Hadoop version 0.16.4 to 0.18.3. Finally, we compiled the

the advices with regioncuts in the old version	12
ones correctly woven	11
ones detected by the assertion	1

Table 4.1. The number of synchronization advices with regioncuts after the update to Hadoop 0.18.3

TaskTracker class with the same aspects and tests to find the aspects that were not correctly woven any more.

Table 4.1 lists the advices after the update. Among 12 advices using a regioncut, 11 advices were correctly woven for the new version of the program. One advice is not correctly woven. The one advice which is not correctly woven is found by assertion for advice. One advice of 11 advices which are correctly woven is applied another method. The synchronization region is moved to another method by extract method refactoring. Since the refactoring does not change behavior of program, assertion for advice does not emit a warning.

4.4 Summary

This chapter introduced an *assertion for advice*. It provides annotations for methods and advice. You can test whether an advice is correctly applied by adding annotations. Assertion for advice partly addresses a risk of fragile pointcut problem in aspect-oriented programming.

In order to address fragile pointcut problem sufficiently, assertion for advice should be improved. This mechanism only detects whether pointcuts or regioncuts select anticipated join points or regions. It cannot detect pointcut or regioncuts that select unanticipated join point or regions. Assertion for advice should be improved to detect join points or regions that are unintentionally selected. Moreover, the @AssertAdvised annotation can be applied to only methods. Therefore, assertion for advice cannot detect a regioncut that selects an unintended region in the same method. Assertion for advice is required to have means to specify regions that are advised with more flexible granularity. Other approaches that do not use syntactic information can be considered. For example, data-flow of resources may be used for detection.

Chapter 5

Method Shelters

This chapter proposes a new module system called *method shelters* [4] to address conflicts among class extensions. This idea is to make some class extensions effective only within the module defining them and ones imported by that module. This system also protects some class extensions from accidental overriding by outer modules, which directly/indirectly import that module. Thus, if programmers carefully control the scope of class extensions, unexpected conflicts among class extensions are avoidable.

Method shelters are designed to provide the local rebinding property but make conflicts avoidable to a certain degree. On the other hand, the refinements of Ruby does not provide the local rebinding property. Classboxes provide it but may cause conflicts among class extensions if multiple versions of class extensions are used in an importing chain.

5.1 Overview

A method shelter, which is a unit of the module system, consists of two *chambers*: an *exposed* chamber and a *hidden* chamber. A chamber contains *import* declarations and method definitions. An import declaration imports another method shelter. A method definition may define a new method added

to an existing class and it may redefine an existing method in an existing class.

Figure 5.1 shows a code sample in Ruby. It is a solution of the problem mentioned in Figure 2.18. In the code in Figure 5.1, three method shelters CoreShelter, AverageShelter, and ClientShelter are defined. CoreShelter has an Integer#div method in its exposed chamber. CoreShelter is imported by ClientShelter in its exposed chamber. Importing another method shelter in an exposed chamber is called *exposedly importing*.

If only exposed chambers are used, method shelters are similar to classboxes. The local rebinding property is preserved. The methods in exposed chambers are executed as if they all were in the exposed chamber of the outermost or root method shelter, which exposedly imports their method shelters directly or indirectly. If there are multiple definitions of the same method m, the method definition of the outermost method shelter S is selected, and other method definitions of m in method shelters imported from S are overridden. Thus, if a method calls another method in the same method shelter, the call selects and executes a different definition of that method in an outer method shelter. Programmers must consider that a method in an exposed chamber may be redefined by another method shelter importing it.

On the other hand, method definitions in a hidden chamber are not visible from the outside. Furthermore, they are never redefined by another method shelter importing them. which contains another Integer#div method. It is visible within AverageShelter but not from ClientShelter, which imports AverageShelter. Thus a call to div at line 31 in ClientShelter never selects the definition in AverageShelter whereas a call to div at line 13 in the exposed chamber of AverageShelter selects the definition at line 20 in the hidden chamber of AverageShelter. The problem in Figure 2.18 does not happen. However, hidden chambers have trade-off. A method defined in a hidden chamber cannot be redefined even if it has a bug and the user wants to fix it by monkey patching. The concept of exposed and hidden chambers are similar to public/private methods in OOP languages. However method shelters are orthogonal to the public/private access control. We decided to use exposed and hidden as keywords to avoid misunderstanding.

Importing another method shelter in a hidden chamber is called *hiddenly* importing. The methods imported in a hidden chamber are visible only within the method shelter importing them, both its exposed and imported chambers. Note that those methods are imported only from an exposed chamber since methods in a hidden chamber are not visible from the outside. The methods imported in the hidden chamber of a method shelter S are not

visible from other method shelters importing S.

The local rebinding property is preserved in a method shelter hiddenly imported. A method imported in a hidden chamber may be redefined in that hidden chamber. A hidden chamber is used to import and redefine several classes freely for local use only.

Our method-shelter system does not allow ambiguity with respect to method lookup. For example, in Figure 5.2, a method shelter S0 imports S1 and S2. Since both S1 and S2 have a method named m in C class, a call to C#m in the method shelter S0 is ambiguous and hence raises an error. It was possible to design the system so that such ambiguity can be implicitly resolved by introducing some precedence rules, for example, the last imported method shelter has the highest precedence. However, we did not adopt such implicit ambiguity resolution since we believe it will confuse programmers.

Global methods. We call methods (re)defined not within a method shelter *global methods*. A method in a method shelter can call a global method. Our module system considers that all global methods are contained in some anonymous method shelter. This method shelter is implicitly exposedly-imported by the method shelter that contains a caller method to a global method. Thus, the global methods can call methods in the exposed chamber where the caller method is defined. A redefinition of a method in that chamber is also effective when a global method calls it. On the other hand, the methods in the hidden chambers of the caller's method shelter are not visible from the global methods. If a global method calls another global method, these two methods can access the same shelters. For example, if a global method m0 is called from a method in a shelter S and m0 calls a global method m1, then m1 can call the same set of methods in the exposed chamber of S that m0 can call.

Entry point. Since a method in a method shelter is not visible from the outside, a special mechanism is needed to call it at the beginning. In other words, we have to jump into a method shelter from normal execution contexts. We call that method shelter *an entry point*, which is the outermost method shelter in the import chain. An appropriate strategy depends on the base language:

- ***Use main function:*** A main function or method can be defined in a method shelter if the base language has it. The method shelter

containing a main function is an entry point. A main function is a function that is first executed when a program starts.

- ***Use special blcok:*** A special code block can be provided for specifying a method shelter. The code block is executed as if it existed within that method shelter. The entry point is that method shelter. Our ruby prototype adopts this strategy since Ruby does not have a main function like other scripting languages.

Note that in our programming model, every library, framework and application program is in a separate method shelter. The method shelter of an application program imports other method shelters of libraries and frameworks. Hence, having an entry point is natural.

5.2 Lookup semantics

This section presents the semantics of method shelters by showing its method lookup algorithm.

5.2.1 Method shelter tree

Method shelters can be imported from other method shelters. Hence the import relation among shelters constructs a directed graph. For the sake of presentation, we first transform this graph into a tree. We will use this tree to describe where we start looking up a method. This transformation is also used in our implementation in Section 5.3 for performance reason.

Figure 5.3 shows an example, where method shelter A imports B and C, and B imports C. We do not have to distinguish a type of importing, exposedly or hiddenly, in this transformation. If a method shelter (C in the example) is imported by multiple different method shelters, the node of that imported method shelter C is duplicated and the importing method shelters B and A import a different node of C (Figure 5.4). The resulting graph after this transformation is a tree, where every (node of a) method shelter is imported by at most one method shelter, that is, every node has at most one parent. We use this property of the tree for describing the algorithm of method lookup. Although this transformation does not work if import relations make a cycle, method shelters prohibit cyclic importing. If cyclic importing is detected, this graph-to-tree transformation raises an error.

Our semantics currently supposes that method shelters are immutable. If importing relations of method shelters are changed at run time, a method shelter tree should be reconstructed.

5.2.2 The lookup algorithm

This section shows the algorithm for looking up a method in a method shelter. Figure 5.5 lists the algorithm written in Scheme. lookup is the main function. It takes three arguments: *context*, *methodname* and *class*. *methodname* and *class* are the name of a called method and the class of the receiver object. *context* is a node in the tree of method shelters mentioned above. It indicates the method shelter that contains the caller method, which is currently running and attempts to call the method on the receiver object. The result of the method lookup depends on where the caller method is located.

lookup first tries to find a method in a given *class* by calling lookup-method-of-class. If a method is not found there, then lookup tries to find a method in the super class. Note that Ruby adopts single inheritance. lookup and lookup-method-of-class return a pair of the found method and the tree node of the method shelter containing that method, which will be implicitly passed to the found method for further method lookup.

lookup-method-of-class looks up a method in method shelters. First, it looks up the hidden chamber of the given method shelter node. If the method is found in that chamber, the found method is returned. If not found, it tries to look up a method again in the subtree rooted at the *source chamber*.

Methods in the given shelter's exposed-side are looked up from the source chamber. The root chamber or hidden chambers which imports the given node can be the source chamber. The one nearest to the given shelter is selected as the source chamber. Figure 5.6 shows the find-source-chamber function that computes the the source chamber.

Figure 5.7 shows an example. Suppose that $S0$ exposedly imports $S1$, $S1$ exposedly imports $S2$, and $S1$ also hiddenly imports $S3$. Then the source chamber of $S0$, $S1$ and $S2$ is the exposed chamber of $S0$. Note that $S0$ is the method shelter at the entry point. The source chamber of $S3$ is the hidden chamber of $S1$ since $S3$ is hiddenly imported.

Figure 5.8 shows the definitions of lookup-exposed, lookup-hidden, and lookup-global functions used in lookup-method-of-class. lookup-exposed first searches the exposed chamber of the given node. If a method is found, the function returns a pair of the given node and the found method body. Otherwise, the function recursively calls itself on all the nodes of the subtree

rooted at the given node although the nodes hiddenly imported are excluded from the search space. Then the function makes a list of the values returned by the recursive calls. The list is processed by filter-methods, which returns an element if the list contains only one element. filter-methods raises an error if the list contains multiple elements since the method to look up is ambiguous. lookup-hidden is similar. It first searches the hidden chamber of the given node and then the subtree rooted at the given node. It searches only the nodes exposedly imported by the given node directly or indirectly. Finally, lookup-global searches the global method table. If it finds a method, it returns a pair of a method-shelter node and the body of the method found. This method-shelter node represents a method shelter that corresponds to the global method table and it is directly imported by the node given to lookup-global.

```
1  shelter :CoreShelter do
2    class Integer
3      def div(x)
4        # <returns integer result>
5      end
6    end
7  end
8
9  shelter :AverageShelter do
10   class Array
11     def avg
12       s = self.sum
13       return s.div(self.size) # rational version is called
14     end
15   end
16
17   hide
18   import :CoreShelter
19   class Integer
20     def div(x)
21       # <returns rational result>
22     end
23   end
24 end
25
26 shelter :ClientShelter do
27   import :Core
28   import :AverageShelter
29   def calc
30     [1,2,3,4].avg # returns "(5/2)"
31     5.div(2) # returns "2"
32   end
33 end
```

Figure 5.1. Code sample (a solution of the problem in Figure 2.18)

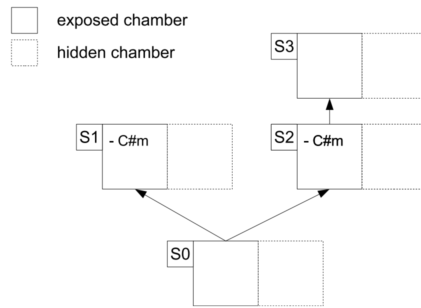


Figure 5.2. Ambiguous methods in a method shelter

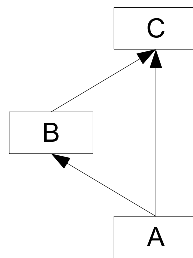


Figure 5.3. An example of an import graph of method shelters

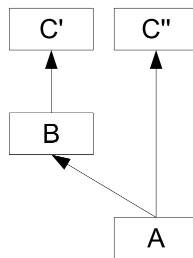


Figure 5.4. A method shelter tree reconstructed from Figure 5.3

```
1  (define (lookup context class methodname)
2    (let ((method (lookup-method-of-class context class methodname
          )))
3      (cond
4        (method method)
5        ((superclass class) (lookup context (superclass class)
              methodname))
6        (else (error "no method error" class name)))))
7
8  (define (lookup-method-of-class context class methodname)
9    (let ((hidden-method (lookup-hidden context class methodname)))
10     (if hidden-method
11         hidden-method
12         (let* ((source-chamber (find-source-chamber context))
13                (source-node (node-of-chamber source-))
14                (exposed-method
15                   (if (is-exposed? chamber)
16                       (lookup-exposed source-node class
                              methodname)
17                       (lookup-hidden source-node class
                              methodname))))
18           (if exposed-method
19               exposed-method
20               (lookup-global node class name))))))
```

Figure 5.5. Method lookup functions of method shelters

```
1  (define (hidden-imported? node)
2    <Is the given node is hidden-imported from parent?>)
3  (define (exposed-imported? node)
4    <Is the given node is exposed-imported from parent>)
5
6  (define (find-source-chamber node)
7    (cond
8      ((not (parent-node node)) (list node 'exposed))
9      ((hidden-imported? node) (list (parent-node node) 'hidden))
10     ((exposed-imported? node) (find-source-chamber (parent-node
           node)))
11   ))
```

Figure 5.6. Definition of source-node and source-chamber

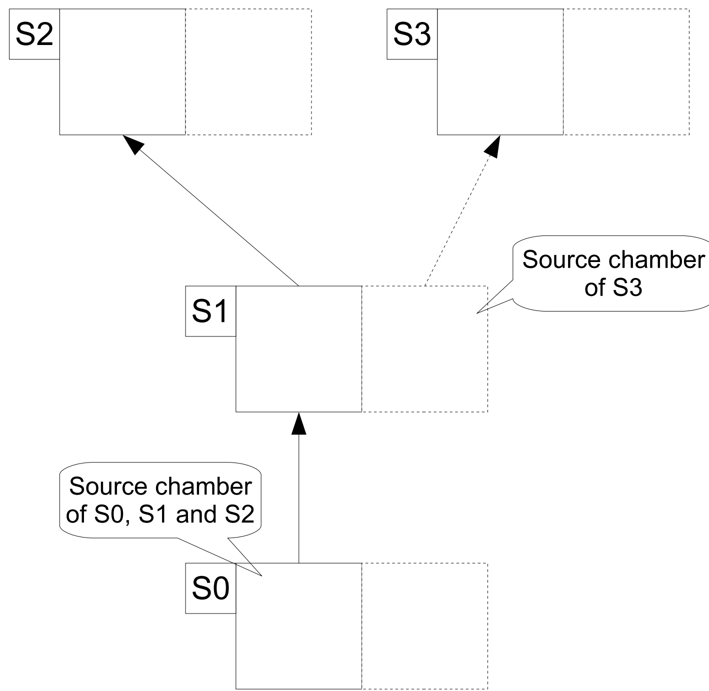


Figure 5.7. An example of source-node and source-chamber

```
1  (define (lookup−exposed node class name)
2    (if (exposed−method−table−exists? node class name)
3        (list node
4              (exposed−method−table−get node class name))
5        (filter−methods class name
6          (map (lambda (e) (lookup−exposed e class name))
7               (exposedly−importings node)))))
8
9  (define (lookup−hidden node class name)
10   (if (hidden−method−table−exists? node class name)
11       (list node
12             (hidden−method−table−get node class name))
13       (filter−methods class name
14         (map (lambda (e) (lookup−exposed e class name))
15              (hiddenly−importings node)))))
16
17 (define (lookup−global node class name)
18   (if (global−method−table−exists? class name)
19     (list <a node which is exposedly imported by the given node>
20           (global−method−table−get table name))
21     #f))
```

Figure 5.8. Definition of lookup-exposed and lookup-hidden

5.3 A proof-of-concept implementation

This section describes a proof-of-concept implementation [1] of method shelters in Ruby since Ruby already has a class extensions feature and its source code is publicly available. We modified the virtual machine of Ruby 1.9.2.

Since Ruby has already powerful expressiveness, we decided not to extend Ruby's syntax. The syntax of method shelters is based on Ruby's syntax. Figure 5.9 shows a sample code for illustrating method shelters' syntax. Although shelter looks like a keyword, it is a method name. "shelter" method takes shelter's name and a block. ":S2" represents a symbol S2. "do ... end" represents a block. The methods defined in the block are contained in the method shelter. By default, those methods belong to an exposed chamber. On the other hand, the methods defined after a call to hide method (at line 15) belong to a hidden chamber. To import another method shelter, call import method. Its argument is a method shelter's name. If import method is called after hide method, the imported shelter is hiddenly imported and hence belongs to a hidden chamber.

The method shelter at the entry point is specified by shelter_eval method. For example, the line 24 and 25 in Figure 5.9 is executed within the contexts of the method shelter S0.

5.3.1 Implementation details

When a shelter method is called, we create a shelter object. A shelter object consists of five members: its name, a list of exposedly imported shelters, a list of hiddenly-imported shelters, an exposed method table and a hidden method table. When a method is defined in a method shelter, its method name is converted to a unique synthesized name. The mapping between the original method name and the converted one is recorded in the shelter object's method table. At method lookup, the table of converted method names is searched first.

As mentioned in Section 5.2, the method lookup algorithm needs a current node in a method-shelter tree representing import relations. To maintain a current node, we added a new member to the stack frame of the Ruby VM (Figure 5.10 shows the definition of the stack frame). When shelter_eval method is called, a method shelter tree is constructed from the specified

[1] The source code is available at
http://github.com/flexfrank/ruby_with_method_shelters

```
1  shelter :S2 do
2    class Integer
3      def inc(n)
4        self + n
5      end
6    end
7  end
8
9  shelter :S1 do
10   class Integer
11     def inc10
12       self.inc(10)
13     end
14   end
15   hide
16   import :S2
17 end
18
19 shelter :S0 do
20   import :S1
21 end
22
23 shelter_eval :S0 do
24   p(1.inc10) # prints 11
25   p(1.inc(1)) # error: method is not found
26 end
```

Figure 5.9. The syntax of method shelters

method shelter and the root node of the tree is set to the stack frame of the block. When a method defined in a shelter is called, a new stack frame is created. The shelter node which the called method is defined is set in the member of the new stack frame.

5.3.2 Optimization

We implemented a few optimization techniques for method shelters to improve execution performance. First, we added a method cache to every node of a method-shelter tree. It records a mapping from a pair of a class name and a method name to a pair of a method entry and the tree node where the method is found. A method entry is a primitive data structure for calling a method in the Ruby VM. A cache entry is updated at method lookup. This

```
1  typedef struct {
2      VALUE *pc;
3      VALUE *sp;
4      VALUE *bp;
5      rb_iseq_t *iseq;
6      VALUE flag;
7      VALUE self;
8      VALUE *lfp;
9      VALUE *dfp;
10     rb_iseq_t *block_iseq;
11     VALUE proc;
12     shelter_node_t* shelter_node; /* current shelter node we added
13     const rb_method_entry_t *me;
14 } rb_control_frame_t;
```

Figure 5.10. The definition of Ruby's stack frame struct.

cache reduces the overhead of method lookup in particular when an import chain is long.

Since method shelters change the algorithm of method lookup, we also modified the implementation of the inline cache of the Ruby VM. The modified implementation records a current node of a method-shelter tree. Figure 5.11 shows the modified definition of struct for inline cache. iseq_inline_cache_entry struct stores inline cache. A pointer to shelter node and shelter_cache_entry are added as members. shelter_node_chache_entry stores information about a method in a shelter. It contains cached method's name, method entry, and the shelter node which the method is defined in.

5.3.3 Compatibility

The implementation of method shelters keeps the compatibility with the original Ruby. A normal Ruby program written without method shelters can run on our modified Ruby interpreter. Although we added an additional member to a stack frame of the Ruby VM, this member for maintaining a current node of a method shelter tree is set to NULL at initialization. If the current node is NULL, the method lookup uses the original algorithm for Ruby.

```
1  typedef struct shelter_node_chache_entry{
2      VALUE vm_state;
3      ID shelter_method_id;
4      rb_method_entry_t* me;
5      shelter_node_t* next_node;
6  } shelter_cache_entry;
7
8  struct iseq_inline_cache_entry {
9      VALUE ic_vmstat;
10     VALUE ic_class;
11     union {
12         VALUE value;
13         struct{
14             void* shelter_node;
15             union{
16                 void* shelter_cache_entry;
17                 rb_method_entry_t *method;
18             }method_e;
19         } method_s;
20         long index;
21     } ic_value;
22 };
```

Figure 5.11. The definition of Ruby's inline cache struct.

5.4 Applications

This section illustrates several examples of the use of method shelters.

5.4.1 Convenient methods in Ruby on Rails

The first example is Ruby on Rails. The ActiveSupport library, which is part of Ruby on Rails, provides a number of convenient methods for Ruby's core classes. Among those methods, we moved time-related methods in the Numeric class into a method shelter. ActiveSupport adds minutes, hour and days methods to Numeric class. These methods return Duration objects representing time. They simplify writing code for calculating time. For example,

```
10.minutes.ago
```

returns Time object representing the time 10 minutes before the current time.

```
1  shelter :ActiveSupportNumericTime do
2   class Numeric
3     # ** snip **
4
5     def days
6       ActiveSupport::Duration.new(self * 24.hours, [[:days, self]])
7     end
8     alias :day :days
9
10    # ** snip **
11  end
12 end
```

Figure 5.12. The time-related methods we defined in a method shelter

```
1  shelter :DateControllerShelter do
2    class DateController < ApplicationController
3      def days_ago
4        @text=params[:id].to_i.days.ago
5      end
6    end
7
8    hide
9    import :ActiveSupportNumericTime
10 end
```

Figure 5.13. A client code of Ruby on Rails

The definitions of these methods can be moved into a method shelter. Figure 5.12 is a code snippet of the method shelter containing these methods. Figure 5.13 shows a controller class for Ruby on Rails. Like a servlet in Java, it is executed when a corresponding web page is accessed by a web browser. This controller class is in a method shelter, which hiddenly imports ActiveSupportNumericTime. Thus, days method in Figure 5.12 is available only in this controller class whereas it is not in the rest of the program. Note that days method is not visible even in method shelters importing the method shelter in Figure 5.13. To call days, method shelters must import ActiveSupportNumericTime again within the method shelters.

5.4.2 Operator redefinition

Section 2.3.3 mentioned a problem with conflicting redefinition of the "/" operator. The sketch of the solution with method shelters was already presented in Figure 5.1.

Figure 5.14 shows a realistic version of the code in Figure 5.1. In Ruby, numbers are represented by Fixnum objects and "/" method is defined in this class. Since the original division method "/" of Fixnum is built in, this code does not include CoreShelter shown in Figure 5.1. The "/" method is redefined in MathNShelter instead of AverageShelter. This simulates Ruby's "mathn" library, which is a separate library providing the redefined "/" method.

Since MathNShelter is hiddenly imported by a method shelter AverageShelter, avg method in Array returns a rational value. The "/" operator at line 13 executes the definition in MathNShelter method shelter. A method shelter ClientShelter can safely import AverageShelter and call avg method without being aware of MathNShelter. Note that since Fixnum is a class in the standard library, calc method can execute the "/" operator at line 25 without explicitly importing Fixnum class. The "/" operator here returns an integer.

Since Ruby is a scripting language, the lines from 28 to 30 compose the code running first when this program is invoked. This "main function" is executed in ClientShelter method shelter.

5.4.3 RSpec

RSpec [2] is a testing tool for Ruby to support Behaviour-Driven Development. We applied method shelters to RSpec.

RSpec can load and test multiple test case files. These files can load other libraries. If a library extends existing classes, one test case may affect other test cases. We modified RSpec to load each test case file in its own method shelter.

Figures 5.15, 5.16, and 5.17 show a simple exmple. "avg.rb" in Figure 5.15 library redefines "/" operator of Fixnum class. The implementation of "/" operator is changed to return the resulst as a Rational obejct. It also adds average method to Array. It returns an average of elements as Rational obejct since it uses "/" operator. Figure 5.16 is a spec file which loads "avg.rb" file and tests average method. "divspec.rb" shown in Figure 5.17 tests "/" operator in Fixnum. The spec file does not consider the operator being redefined. Although the spec files succeed when you run them independently, "divspec.rb" fails if you test them at the same time on the original Rspec.

The modified RSpec evaluates a spec file in a dependent method shelter. The modified Ruby interpreter with method shelters introduces load_in_shelter method. The method loads a source file in current method shelter definition. The scope of the method definition in the loaded files are restricted in the method shelter. In the example, redefinition of "/" is only available in "avgspec.rb" if you use load_in_shelter instead of require.

5.4.4 Protecting optimized methods

The Ruby VM optimizes several special methods including arithmetic operators. When one of the special methods is called and it is not redefined by the users, the VM directly performs its operation instead of executing that method. The VM manages for every operator a flag indicating whether or not the special methods are redefined. The receiver class is not considered for a reason of performance trade-off. Thus, if "+" operator for Integer is redefined by the users, the VM recognizes all "+" operators including one for Float are also redefined and makes them unoptimized. Redefining a single special method may cause serious performance overhead.

If such a special method is redefined in a method shelter, the VM can directly perform the optimized operation when it is out of that method shelter. The implementation of method shelters manages the flags per method shelter. Hence, if a method shelter S redefines a special method in a hidden chamber, that redefinition is not visible from other method shelters importing S and the VM performs optimized operations for special methods in these method shelters. Otherwise, if a method shelter S_1 redefines a special method in an exposed chamber and another method shelter S_2 hiddenly imports S_1 for reusing the redefinition, then the redefinition is not visible from method shelters importing S_2, which are ones indirectly importing S_1. The VM performs optimized operations in these method shelters.

5.4.5 Private instance variables

In Ruby, private instance variables are not available. A method shelter can be used to define private instance variables visible only within the method shelter.

Figure 5.18 shows the code for defining getter and setter methods for accessing an instance variable with a newly generated unique name. When shelter_accessor method is called, accessor methods with the given name are defined. Note that in Ruby an instance variable is automatically created

when it is first used. The code in Figure 5.18 does not use method shelters but the reflection capability of Ruby. get_var_name_for_current_shelter returns a unique name for the given name and the caller's method shelter. If the name and the shelter are same it returns the same variable name.

Figure 5.19 shows the client code. Two method shelters S0 and S1 add accessor methods to Object class. Although both the names of the added instance variables are counter, they access different instance variables. The methods defined by a call to shelter_accessor in different method shelters are distinct.

```
 1  shelter :MathNShelter do
 2    class Fixnum # fixed size integer in Ruby
 3      def /(x)
 4        Rational(self,x)
 5      end
 6    end
 7  end
 8
 9  shelter :AverageShelter do
10    class Array
11      def avg
12        sum = self.inject(0){|r,i|r+i}
13        sum / self.size
14      end
15    end
16    hide
17    import :MathNShelter
18  end
19
20  shelter :ClientShelter do
21    import :AverageShelter
22
23    def calc
24      p([1,2,3,4,5,6,7,8,9,10].avg) # prints "(11/2)"
25      p(55/10) # prints 5
26    end
27  end
28
29  shelter_eval :ClientShelter do
30    calc
31  end
```

Figure 5.14. The code that redefines "/" methods in method shelters

```ruby
class Fixnum
  def /(o)
    Rational(self,o)
  end
end
class Array
  def average
    self.inject(&:+)/self.size
  end
end
```

Figure 5.15. "avg.rb" library that redefines / and adds average method to Array class

```ruby
require "./avg.rb"
#load_in_shelter "./avg.rb"

describe Array, "when empty" do
  it "should be 5" do
    [1,2,3,4,5,6,7,8,9,10].average.should == Rational(11,2)
  end
end
```

Figure 5.16. "avgspec.rb" spec file for average method written in RSpec

```ruby
describe Integer do
  it "should be 0" do
    (1/2).should == 0
  end
end
```

Figure 5.17. "divspec.rb" for Fixnum's / written in RSpec

```
1  class Module
2    def shelter_accessor(name)
3      define_method name do
4        ivname= get_var_name_for_current_shelter(name)
5        self.instance_variable_get(ivname)
6      end
7
8      define_method (name.to_s+"=").to_sym do|val|
9        ivname= get_var_name_for_current_shelter(name)
10       self.instance_variable_set(ivname,val)
11     end
12   end
13 end
```

Figure 5.18. The code for defining getter and setter methods to access a private instance variable

```
1  shelter :S0 do
2    class Object
3      shelter_accessor :counter
4    end
5  end
6  shelter :S1 do
7    class Object
8      shelter_accessor :counter
9    end
10 end
11
12 o=Object.new
13 shelter_eval :S0 do
14   o.counter=0
15   p o.counter #prints 0
16 end
17 shelter_eval :S1 do
18   p o.counter #prints nil
19   o.counter=1
20   p o.counter #prints 1
21 end
22 shelter_eval :S0 do
23   p o.counter #prints 0
24 end
```

Figure 5.19. The client code using accessor methods to a private instance variable

	Avg. time (s)	SD[3]
On the original VM	1.430	0.010
On our VM without method shelters	1.575	0.018
With 1 method shelter	1.476	0.013
With 5 method shelters	1.493	0.018

Table 5.1. Execution time of empty method (1,000 tries)

5.5 Performance

In this section, we discuss the performance of our prototype implementation of method shelters. The implementation is based on Ruby 1.9.2 [2]. We compare it with the original implementation of Ruby 1.9.2. We ran our benchmark programs on Mac OS X 10.6 with 2.54GHz Intel Core 2 Duo processor and 4GB memory.

5.5.1 Micro benchmark

First, to measure an overhead of method lookup, we ran a program that calls a method with an empty body. The benchmark program calls an empty method 10,000,000 times. We prepared five environments: the original Ruby VM, our modified VM without method shelters, our VM with one method shelter and our VM with five method shelters imported. The benchmark code with five method shelters is shown in Figure 5.20. We ran the benchmark programs 1,000 times on each environment.

Table 5.1 shows the results. When method shelters are not used, our VM runs 10% slower than the original VM. This is because our VM must check whether a method shelter is passed or not on method lookup. When one method shelter is used, the overhead is about 3%. Method shelters make method lookup faster, this is due to method caches that we added. When five method shelters are used, it works with comparative speed to one method shelter. This result is also due to the caches.

We also measured execution time of the Fibonacci function under the same environments as above. Table 5.2 lists the results. In this case the overhead of our VM is about 14% and with method shelters is 18% to 19%.

[2] The revision number of Ruby's subversion repository is 30579

[3] standard deviation

```
1  shelter :S0 do
2    def a
3    end
4  end
5  shelter :S1 do import :S0 end
6  shelter :S2 do import :S1 end
7  shelter :S3 do import :S2 end
8  shelter :S4 do import :S3 end
9
10 shelter_eval :S4 do
11   10000000.times do
12     a
13   end
14 end
```

Figure 5.20. The benchmark program that calls an empty method under five method shelters

	Avg. time (s)	**SD**
On the original VM	1.000	0.005
On our VM without method shelters	1.141	0.004
With 1 method shelter	1.180	0.036
With 5 method shelters	1.192	0.049

Table 5.2. Execution time of fib(33) (1,000 trials)

5.5.2 tDiary

To measure the performance of method shelters on a real application, we applied method shelters to *tDiary* [53], a web-based diary system written in Ruby. We used tDiary 3.0.1 for this benchmark. tDiary 3.0.1 redefines three methods in String class: to_a, each and method_missing. We redefined these three methods in a method shelter and ran the main code of tDiary in a method shelter importing it. We ran tDiary on Apache 2.2.17 with CGI and measured response time by ApacheBench. For comparison, we used three versions of tDiaries: tDiary without method shelters on the original Ruby VM, without method shelters on our Ruby VM and with method shelters on our Ruby VM. We accessed the top page of each diary 300 times.

Table 5.3 lists the results. This results show that our modified VM does not impact performance of existing applications when method shelters are

	Avg. time (ms)	SD
On the original VM	704	7.1
On our VM without method shelters	704	6.6
With method shelters	627	6.5

Table 5.3. Response time of tDiary (300 trials)

not used. It also indicates method shelters improve the execution speed. This is due to Ruby VM's optimizations that we mentioned in Section 5.4.4. method_missing, which we confined into a method shelter, is the one of special methods. method_missing is a hook method that is called when an undefined method is called. If method_missing is not redefined, the VM can skip a call to it since the default definition is empty. In this benchmark, we redefined method_missing for String in a method shelter. Hence this redefinition does not affect the performance of the code out of that method shelter. On the other hand, tDiary running on the original Ruby VM gets performance penalties due to the redefinition of method_missing. This is why method shelters improved the execution performance of this benchmark test.

5.5.3 Ruby on Rails

We applied method shelters to Ruby on Rails in Section 5.4.1. We measured the performance of a Ruby on Rails application with method shelters. Figure 5.21 is a benchmark program we used. index method is an action method, which calculates time and accesses a database once. We used SQLite 3.6.12 for a database engine. The version of Ruby on Rails is 3.0.7. We ran this application on WEBrick, a web server written in Ruby. We requested the action 1,000 times through ApacheBench and measured response time.

Table 5.4 lists the results in *development environment*. In this environment, user-defined application classes are reloaded per request. In this case, method shelters made the execution performance about 50% slower. Table 5.5 lists the results in *production environment*, in which application classes are not reloaded per request. In this environment, the overhead is less than 4%. This difference between two environments result from the hit ratio of method caches. In the development environment, whenever classes are reloaded, the VM invalidates method caches for method shelters. This implies serious performance penalties.

Table 5.6 lists the hit ratio of method caches in method shelters after

```
1  class TestController < ApplicationController
2    def index
3      @text="#{(1.day.ago + 1.day)}"
4      @accesses=Access.order("id desc").limit(10).find_all.to_a.inspect
5    end
6  end
```

Figure 5.21. The benchmark program for Ruby on Rail

	Avg. time (ms)	SD
On the original VM	53.131	14.7
On our VM without method shelters	53.341	14.7
With method shelters	78.871	16.2

Table 5.4. Response time of Rails application (1,000 trials, development env.)

warming-up. In the production environment, over 90% and 100% of lookups hit inline method cache. In the development environment, less than 75% hit inline caches. This result indicates that method shelter is not so slow when method caches are appropriately filled.

5.6 Summary

This chapter presents *method shelters* to control scope of class extensions. A method shelter is a module that restricts scope of class extensions. A method shelter has an exposed chamber for public API and a hidden chamber for internally used methods. Programmers can avoid conflicts among class extensions by composing modules with these scopes.

	Avg. time (ms)	SD
On the original VM	10.865	7.7
On our VM without method shelters	11.049	7.8
With method shelters	11.296	7.7

Table 5.5. Response time of Rails application (1,000 trials, production env.)

	development	production
Inline cache hit (%)	58.35	92.55
Total cache hit (%)	74.0	100.0

Table 5.6. Cache hit ratios of Rails application (1,000 trials, production env.)

Chapter 6

Conclusion

This thesis has discussed replacement of program according to situations with destructive extensions. The language constructs presented in this thesis address problems with destructive extensions: expressiveness, fragile pointcut problem, and interference among extensions. These constructs help to replace programs without modifying existing source code in modular way. Regioncut introduces means to select code regions as join points. It enables us to separate synchronization concerns into an aspect. Assertion for advice provides means to guarantee that an advice is correctly applied by annotating methods and advice. Method shelters enable us to control scope of class extensions with two types of scope: exposed chambers and hidden chambers. Programmer can avoid interference of class extensions by composing modules with these two scopes.

Contributions

The contributions by this thesis are summarized as follows:

- This thesis introduced a new languages construct to deal with code regions in aspect-oriented programming. In existing AOP languages, you can change behavior of join points: an expression or a method

body. The granularity of join points is not suitable to separate several concerns including synchronization into aspects. This thesis shows that code regions, which have granularity between an expression and a method body, can be treated as join points and that regioncuts improve expressiveness of AOP.

- Then, this thesis proposes a new language construct to guarantee that the mandatory concerns written in an aspect are correctly applied. Existing AOP languages do not notify developers whether an advice is applied or not. This construct helps to address fragile pointcut problem in aspect-oriented programming languages.

- This thesis also presents a module system to control scope of class extensions. In existing programming languages with class extensions, multiple class extensions may be in conflict, and it is difficult to control and avoid conflicts. This module system introduces two scopes of class extension. By combining these scopes, programmers can use multiple class extensions in one program and avoid interferences or conflicts among class extensions.

Future Directions

Possible future directions of this thesis are as follows:

Exploring large-scale case studies for regioncut and assertion for advice Regioncut is applied to Javassist, and regioncut and assertion for advice is applied to one class in Hadoop. Although these programs are practical, case studies shown in this thesis is small. For example, they should be applied to all classes of Hadoop or other large-scale software.

Applying method shelters to pointcut/advice This thesis introduced method shelters into class extensions. Method shelters can control scope of extensions; they can be applied to pointcut/advice mechanism in aspect-oriented programming. Method shelters for pointcut/advice can avoid interference among pointcuts/advice. Destructive extensions, which include pointcut/advice and class extensions, can get less interference among extensions.

Applying method shelters to static programming languages Methods shelters are implemented in Ruby programming language, which is dynamically-typed and permits us to extend programs destructively at runtime. This mechanism should be applied to statically-typed programming languages including Java. To implement method shelters in statically-typed languages, relationships of shelters should be determined at compile time. Furthermore, which method definition is called should be determined at compile-time for better performance. To apply method shelters to statically-typed languages, the semantics of method shelters should be formalized. Moreover, whether method shelters preserve soundness of a type system or not should be proved.

Simplifying method shelters Method shelters should be brushed up in order to be adopted by real programming languages. Method shelters have room for simplification. For example, hidden chambers can be eliminated. Features of method shelters can be fulfilled by using exposed chambers and hiddenly-imports. Simplification helps to describe semantics of method shelters formally. Method shelters will be more comprehensible by simplification. It may increases a possibility that method shelters will be adopted by real programming languages.

Bibliography

[1] . [#jassist-28] javassist enhancement failed on deserializing hibernate proxies - jboss.org jira. `http://jira.jboss.org/jira/browse/JASSIST-28`.

[2] . Rspec.info: home. http://rspec.info/, 2012.

[3] Shumpei Akai and Shigeru Chiba. Extending aspectj for separating regions. In *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, pages 45–54, New York, NY, USA, 2009. ACM.

[4] Shumpei Akai and Shigeru Chiba. Method shelters: avoiding conflicts among class extensions caused by local rebinding. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 131–142, New York, NY, USA, 2012. ACM.

[5] Shumpei Akai, Shigeru Chiba, and Muga Nishizawa. Region pointcut for aspectj. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '09, pages 43–48, New York, NY, USA, 2009. ACM.

[6] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 39–50, New York, NY, USA, 2009. ACM.

[7] Jonathan Aldrich. Open modules: modular reasoning about advice. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 144–168, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM.

[9] Sven Apel and Don Batory. When to use features and aspects?: a case study. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 59–68, New York, NY, USA, 2006. ACM.

[10] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[11] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 177–189, New York, NY, USA, 2005. ACM.

[13] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, October 2005.

[14] Gilad Bracha, Peter Ah, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In Theo D 'Hondt,

editor, *ECOOP 2010 — Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer Berlin Heidelberg, 2010.

[15] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, pages 303–311. ACM, 1990.

[16] Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. Ejflow: taming exceptional control flows in aspect-oriented programming. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 72–83, New York, NY, USA, 2008. ACM.

[17] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, London, UK, 2000. Springer-Verlag.

[18] Shigeru Chiba, Atsushi Igarashi, and Salikh Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 539–554, New York, NY, USA, 2010. ACM.

[19] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[21] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, June 2005.

[22] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[23] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

[24] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[25] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *Software, IEEE*, 23(1):51 –60, jan.-feb. 2006.

[26] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM.

[27] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 411–428, New York, NY, USA, 1993. ACM.

[28] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 85–95, New York, NY, USA, 2007. ACM.

[29] Grel Hedin and Eva Magnusson. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37 – 58, 2003.

[30] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[31] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[32] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[33] Benoit Kessler and Éric Tanter. Analyzing interactions of structural aspects. In *Workshop on Aspects, Dependencies and Interactions @ECOOP 2006*. Springer Berlin Heidelberg, 2006.

[34] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01*, pages 327–353. Springer-Verlag, 2001.

[35] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akit and Satoshi Matsuoka, editors, *ECOOP'97 —Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[36] Karl Klose and Klaus Ostermann. Back to the future: Pointcuts as predicates over traces. In *FOAL 2005: Foundations of Aspect-Oriented Languages 2005*. ACM, 2005.

[37] Kenichi Kourai, Hideaki Hibino, and Shigeru Chiba. Aspect-oriented application-level scheduling for j2ee servers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 1–13, New York, NY, USA, 2007. ACM.

[38] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the beta programming language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 285–298, New York, NY, USA, 1983. ACM.

[39] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 36–44, New York, NY, USA, 1998. ACM.

[40] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM.

[41] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In *FOAL 2002: Foundations Of Aspect-Oriented Languages - Workshop at AOSD 2002 -*. ACM, 2002.

[42] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[43] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to aspectj. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 39–50, New York, NY, USA, 2006. ACM.

[44] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[45] D.L. Parnas. Software engineering or methods for the multi-person construction of multi-version programs. In ClemensE. Hackl, editor, *Programming Methodology*, volume 23 of *Lecture Notes in Computer Science*, pages 225–235. Springer Berlin Heidelberg, 1975.

[46] Rails core team. Ruby on rails. http://rubyonrails.org/, 2011.

[47] André Restivo and Ademar Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. In *Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*, SPLAT '07, New York, NY, USA, 2007. ACM.

[48] Ruby community. Ruby programming language. http://www.ruby-lang.org/, 2011.

[49] Hossein Sadat-Mohtasham and H. James Hoover. Transactional pointcuts: designation reification and advice of interrelated join points. In *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, pages 35–44, New York, NY, USA, 2009. ACM.

[50] Kouhei Sakurai and Hidehiko Masuhara. Test-based pointcuts for robust and fine-grained join point specification. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 96–107, New York, NY, USA, 2008. ACM.

[51] Maximilian Stoerzer and Juergen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings*

of the 21st IEEE International Conference on Software Maintenance, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Fuminobu Takeyama and Shigeru Chiba. An advice for advice composition in aspectj. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 122–137, Berlin, Heidelberg, 2010. Springer-Verlag.

[53] tDiary.org. tDiary. http://sourceforge.net/projects/tdiary/, 2011.

[54] The Apache Software Foundation. Welcome to apache hadoop! `http://hadoop.apache.org/`.

[55] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[56] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[57] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 159–169, New York, NY, USA, 2004. ACM.

[58] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. Javagi: generalized interfaces for java. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 347–372, Berlin, Heidelberg, 2007. Springer-Verlag.

[59] Stefan Wehr and Peter Thiemann. Javagi: The interaction of type classes with interfaces and inheritance. *ACM Trans. Program. Lang. Syst.*, 33(4):12:1–12:83, July 2011.

[60] Allen Wirfs-Brock and Brian Wilkerson. A overview of modular smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '88, pages 123–134, New York, NY, USA, 1988. ACM.

[61] Chenchen Xi, Bruno Harbulot, and John R. Gurd. A synchronized block join point for aspectj. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 39–39, New York, NY, USA, 2008. ACM.

[62] Chenchen Xi, Bruno Harbulot, and John R. Gurd. Aspect-oriented support for synchronization in parallel computing. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution*, pages 1–5, New York, NY, USA, 2009. ACM.