ASPECT IMPACT ANALYSIS

by

Dehua Zhang

School of Computer Science
McGill University, Montréal

August 2008

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2008 by Dehua Zhang

Abstract

One of the major challenges in aspect-oriented programming is that aspects may have unintended impacts on a base program. Thus, it is important to develop techniques and tools that can both summarize the impacts and provide information about the causes of the impacts. This thesis presents impact analyses for AspectJ.

Our approach detects different ways advice and inter-type declarations interact and interfere with the base program and focuses on four kinds of impacts, *state impacts* which cause changes of state in the base program, *computation impacts* which cause changes in functionality by adding, removing or replacing computations of the base program, *shadowing impacts* which cause changes of field reference in the base program, and *lookup impacts* which cause changes of method lookup in the base program.

We provide a classification scheme for these kinds of impacts and then develop a set of static analyses to estimate these impacts. A key feature of our approach is the use of points-to analysis to provide more accurate estimates. Further, our analysis results allow us to trace back to find the causes of the impacts.

We have implemented our techniques in the AspectBench compiler. By implementing them in an AspectJ compiler, all kinds of pointcuts, advice and inter-type declarations can be analyzed. We also have integrated these analyses into an AspectJ IDE and provided a two-way navigation between impacts and program source code. In addition, we have carried out experiments on example programs and benchmarks to investigate the results of our analyses.

Résumé

L'un des principaux défis de la programmation orientée aspect est que les aspects peuvent avoir des effets non voulus sur le programme de base. Il est donc important de développer des techniques et des outils qui peuvent mesurer les impacts et fournir des informations sur ce genre de phénomène. Cette thèse présente des analyses d'impact pour AspectJ.

Notre approche examines les différentes voies par lesquelles les aspects peuvent interagir avec le programme de base et se concentre sur quatre types d'impacts, *les impacts d'état* qui provoquent des changements d'état dans le programme de base, *les impacts de calcul* qui provoquent des changements au niveau des fonctions par l'ajout, la suppression ou le remplacement de calculs le programme de base, *les impacts d'ombres* qui provoquent des changements de domaine références dans le programme de base et *les impacts de référence* qui provoquent des changements au niveau des méthodes référencées.

Nous offrons un systéme de classification pour ces types d'impacts et développons une série d'analyses statiques pour évaluer ces impacts. Un élément clé de notre approche est l'utilisation d'analyses de pointeurs afin de fournir des estimations plus précises. En outre, nos résultats d'analyse nous permettent de remonter plus loin et de trouver les causes de ces impacts.

Nous avons mis en place nos techniques dans le compilateur AspectBench pour AspectJ. En les appliquant dans un compilateur AspectJ, plusieurs sortes de déclarations peuvent être analysées. Nous avons également intégré ces des analyses dans un environnement de développement AspectJ et avons fourni une navigation bidirectionnelle entre les impacts et le code source. En outre, nous avons procédé à des expériences sur des programmes de test pour démontrer les résultats d'analyses typiques.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Laurie Hendren for guiding and encouraging me to choose this interesting topic, for sharing her ideas, making suggestions and providing assistance throughout the research and writing of this thesis. This work would not have been possible without the support and encouragement of her.

I would also like to thank fellow graduate student Eric Bodden as well as the rest of the Sable Research Group. Additional thanks to Maxime Chevalier-Boisvert for translating the thesis abstract into French.

This work was mainly funded by the NSERC and FQRNT, and I am thankful to those committee members for recognizing and supporting computer science research.

Finally I would like to thank my parents and my sister for their support and my wife for believing in me and putting up with me throughout this very challenging time.

Table of Contents

Abstract **i**

Résumé **iii**

Acknowledgements **v**

Table of Contents **vii**

List of Figures **xi**

List of Tables **xiii**

List of Listings **xv**

List of Listings **xv**

List of Algorithms **xvii**

1 Introduction and Motivation **1**

1.1 Motivation 1

1.2 Contribution 2

1.2.1 Classification of Impacts 2

1.2.2 Static Analyses in the AspectBench Compiler 3

1.2.3 IDE Integration 3

1.2.4 Experiments 4

1.3 Organization 4

2 Background 5
2.1 Tools Overview 5
2.1.1 Soot 5
2.1.2 abc 8
2.1.3 Eclipse 12
2.1.4 AJDT 12
2.2 The Benchmarks 13

3 Four Kinds of Impact 15
3.1 Overview 15
3.2 Examples 16
3.2.1 Bank 16
3.2.2 Source Code Repository 21
3.3 State Impact 24
3.4 Computation Impact 25
3.4.1 Exact-proceed 25
3.4.2 Invariant Advice 26
3.4.3 Variant Advice 26
3.5 Shadowing Impact 28
3.6 Lookup Impact 30
3.7 Aspect Interference 32
3.7.1 Java part is the base program 32
3.7.2 Everything except me 32

4 State Impact 35
4.1 Definition 35
4.2 Analysis 36
4.2.1 Direct State Impact 36
4.2.2 Indirect State Impact 38
4.2.3 Distinguish Direct and Indirect State Impact 39
4.3 Experimental Results 40

4.3.1 Examples 40
4.3.2 Benchmarks 42

5 Computation Impact **45**
5.1 Definition 45
5.2 Analysis 46
5.2.1 Exact-proceed Analysis 47
5.2.2 Computation Impact 56
5.3 Experimental Results 60
5.3.1 Examples 60
5.3.2 Benchmarks 61

6 Shadowing Impact **65**
6.1 Inter-type Field Declaration 66
6.1.1 Definition 66
6.1.2 Analysis 67
6.2 Inter-type Parents Declaration 68
6.2.1 Inter-type-extends-declaration 69
6.2.2 Inter-type-implements-declaration 73
6.3 Experimental Results 73
6.3.1 Examples 73
6.3.2 Benchmarks 74

7 Lookup Impact **77**
7.1 Finding the Matched Method 77
7.1.1 Accessible Methods and Invocation Place 78
7.1.2 Applicable Methods 79
7.1.3 Most Specific Method 80
7.2 Inter-type Method Declaration 81
7.2.1 Definition 81
7.2.2 Analysis 82
7.3 Inter-type Constructor Declaration 85

7.3.1 Definition . . . 85
7.3.2 Analysis . . . 85
7.4 Inter-type Parents Declaration . . . 87
7.4.1 Inter-type-extends-declaration . . . 87
7.4.2 Inter-type-implements-declaration . . . 90
7.5 Experimental Results . . . 90
7.5.1 Examples . . . 90
7.6 Benchmarks . . . 93

8 Visualization - Eclipse Plug-in **95**
8.1 Overview . . . 95
8.2 Running **AIA** . . . 96
8.3 Impact View . . . 98
8.4 Impact Marker . . . 104
8.5 Summary . . . 104

9 Related Work **107**
9.1 Analyzing, Categorizing and Classifying Aspects . . . 107
9.2 Improving AOP Language and Enhancing Reasoning . . . 109

10 Conclusions and Future Work **113**
10.1 Conclusions . . . 113
10.2 Future Work . . . 114

Bibliography **117**

List of Figures

2.1 abc overall design, adapted from [dM04] 9
2.2 Code generation and static weaving of abc, and the pre-weave hierarchy recording phase of **AIA** in static weaving, extended from [ACH+05] 10
2.3 Advice weaving and post-processing of abc [dM04] 11
2.4 **AIA** in post-processing 11

3.1 Class hierarchy of account classes 18
3.2 Class hierarchy of credit card classes after applying `CreditCardAspect` . 21
3.3 Comparison of class hierarchy of credit card classes with and without `GoldCard.bonus` inter-type declaration 28
3.4 Comparison of class hierarchy of credit card classes with and without `ValueCard extends RewardCard` inter-type declaration 30
3.5 Comparison of class hierarchy of credit card classes with and without `GoldCard.payment()` inter-type declaration 31

6.1 Shadowing impact analysis on inter-type field declaration example 69
6.2 An example class hierarchy 69
6.3 Code generation and static weaving of abc, and the pre-weave hierarchy recording phase of **AIA** in static weaving, extended from [ACH+05] 72
6.4 Two inter-type-extends-declarations on the same type example 72
6.5 Inter-type implements declarations example 73

7.1 Invocation place example 79
7.2 Lookup impact analysis on inter-type method declaration example 82
7.3 Lookup impact analysis on inter-type constructor declaration example . . . 85

7.4 Inter-type-extends-declarations example 89

8.1 Eclipse plug-in snapshot 96
8.2 Main class selection dialog 97
8.3 Running progress dialog 97
8.4 Advice highlighting 99
8.5 Impact view displaying computation and state impacts 100
8.6 Impact view displaying shadowing and lookup impacts 102
8.7 Impact view linking with the editor 103

List of Tables

2.1 Basic benchmark metrics 14

3.1 Classification of computation impacts 27

4.1 Basic statistics about benchmarks 43

5.1 Basic statistics about benchmarks 62

6.1 Basic statistics about benchmarks 75

7.1 Analysis result of inter-type method declaration example 84
7.2 Basic statistics about benchmarks 94

List of Listings

2.1 stepPoly in Java form [VRHS+99] . . . 6
2.2 stepPoly in Jimple form [VRHS+99] . . . 7
2.3 stepPoly in disassembled bytecode form . . . 8
3.1 AbstractAccount.java . . . 17
3.2 CheckingAccount.java . . . 17
3.3 StudentCheckingAccount.java . . . 17
3.4 AccountAspect.aj . . . 18
3.5 AbstractCreditCard.java . . . 19
3.6 RewardCard.java . . . 19
3.7 ValueCard.java . . . 19
3.8 GoldCard.java . . . 20
3.9 CreditCardAspect.aj . . . 20
3.10 Bank.java . . . 22
3.11 Output of bank example . . . 22
3.12 SourceCodeRepository.java . . . 23
3.13 SourceCodeRepositoryAspect.aj . . . 24
4.1 Report of analyzing advice in the bank example . . . 41
4.2 Report of analyzing the glass and table example . . . 42
5.1 UnchangedParamsAnalysis example . . . 50
5.2 UnchangedReturnAnalysis example . . . 53
5.3 UnchangedReturnAnalysis example demonstrating side-effect . . . 54
5.4 Report of analyzing advice in the bank example . . . 61
5.5 Report of analyzing the source code repository example . . . 61

6.1 Report of analyzing inter-type declarations in the bank example 74
7.1 Report of analyzing inter-type declarations in bank example 91
7.2 Report of analyzing the source code repository example 93

List of Algorithms

4.1 State impact analysis 36
4.2 Analyze(body, adstmt) in Algorithm 4.1 37
5.1 Computation impact analysis 47
6.1 Shadowing impact analysis on inter-type field declaration 67
7.1 Lookup impact analysis on inter-type method declaration 83
7.2 Lookup impact analysis on inter-type constructor declaration 86

Chapter 1
Introduction and Motivation

1.1 Motivation

Aspect-oriented programming (**AOP**) introduces aspects as language constructs that address cross-cutting concerns [SW07]. Aspects can observe, alter or augment the behavior of base programs. Although this functionality is very powerful, it is also possible that aspects break encapsulation or impact on the based program in unintended ways. The purpose of this work is to provide static analyses that can summarize impacts and help programmers understand impacts and locate the causes of impacts in AspectJ programs.

AspectJ is a popular AOP language which is defined as a convenient extension of Java. An aspect defined in AspectJ can declare different kinds of advice and inter-type declarations. Advice can modify the base program state by writing to fields or change the program's execution by adding to, substituting, repeating or eliminating computations. Inter-type declarations can interfere with field reference and method lookup by introducing new members or modifying the class hierarchy in base program. Complex interactions between the base program and aspects can make AspectJ programs difficult to understand and maintain. The possibility of obliviously and globally changing the behavior of the base program [Stö03b] may lead to undesired and unexpected impacts. In particular, AspectJ users may find that aspects interfere or interact with classes, components or data structures in a way that was not anticipated. Thus, even though many programmers have good uses for aspects,

the uncertainty about the impacts of the aspects on the base code can limit adoption of AspectJ. Therefore, techniques and tools that can both analyze impacts of aspects on base programs and summarize the causes of impacts are desired. In addition, most programmers are accustomed to developing their software using an Integrated Development Environment (**IDE**), thus integrating the analysis results into an IDE and providing a graphical/interactive presentation of analysis results is desired.

1.2 Contribution

To address theses issues, we studied the complicated interactions between aspects and base programs in AspectJ and proposed a concise classification of impacts based on state and computation changes caused by advice and inter-type declarations. We also developed a set of static analysis to analyze these impacts. Finally, we designed and implemented an Eclipse plug-in to present analysis results graphically and interactively. We refer to our collection of tools and techniques as **AIA**— Aspect Impact Analysis.

1.2.1 Classification of Impacts

Since the behavior of a program depends on both the program state and computations executed, and an aspect interferes or interacts with the base program mainly through advice and inter-type declarations, we classify impacts into four categories.

state impact: Indicates changes of state in the base program caused by advice interacting with base program.

computation impact: Indicates changes to which computations are performed caused by advice interacting with base program.

shadowing impact: Indicates changes of field references in the base program caused by inter-type declarations interfering the base program.

lookup impact: Indicates changes of method lookups in the base program caused by inter-type declarations interfering the base program.

We further classify state impact into *direct state impact* which indicates field changes caused by code in the advice body directly, and *indirect state impact* which indicates field changes caused by methods called or transitively called by code in the advice body. In addition, we further classify computation impacts into *invariant* or *variant*. We expect most advice to actually do something, so most advice will have a variant computation impact. Variant computation impacts come in five flavors, *addition*, *elimination*, *definite-substitution* and *conditional-substitution* and *mixed* based on whether computations were added, eliminated, definitely substituted, conditionally substituted, or a combination of different possibilities.

1.2.2 Static Analyses in the AspectBench Compiler

Following the above classification, we implemented a series of static analyses in the back-end of the AspectBench compiler, abc, to analyze the impacts of aspects on both the state and computation of a base program so that the hidden impacts of aspects are revealed. We analyze how fields of base classes are accessed to expose state impacts; we analyze the effect of advice on computation by categorizing each advice into the appropriate computation impact category; we analyze how field references are changed in a base program to expose shadowing impacts; and we analyze how method lookups are changed in a base program to expose lookup impact. Key features of our approach are that we make use of the points-to analysis available in abc to give a more precise analysis, and we use analysis results to provide a more descriptive and informative analyses report, which provides both information about the impacts and the causes of impacts which can guide the programmer in understanding these impacts.

1.2.3 IDE Integration

We also integrated the analyses into an IDE. We implemented an Eclipse plug-in to run these analysis under the most popular AspectJ IDE — AspectJ Development Tools (**`AJDT`**), which is an Eclipse plug-in itself. We visualize analysis results and provide programmers a two-way navigation between the analysis results and the source code. Code involved in impacts are marked in the source code editor so that programmers can navigate from the

source code to the analysis results; and the presentation of analysis results is interactive so that programmers can navigate directly to the parts of the source code involved in the impacts.

1.2.4 Experiments

We experimented with four example AspectJ programs and eight benchmarks to illustrate how our classification and analyses can help programmers understand AspectJ programs and even fix bugs caused by improperly designed advice. We also present screen shots of analyzing example AspectJ programs in the Eclipse plug-in.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 reviews the background tools, including Soot, abc, Eclipse and AJDT, upon which this work is based. Chapter 3 introduces four different kinds of impact and provides some motivating examples. Chapters 4, 5, 6 and 7 discuss *state impact*, *computation impact*, *shadowing impact*, and *lookup impact* in detail and present the corresponding static analysis to discover these impacts, respectively; experimental results of examples and benchmarks for each kind of impact are also given in each chapter. Chapter 8 introduces the Eclipse plug-in and discusses its usage and implementation in detail. Chapter 9 discusses related work. Finally, in Chapter 10, we give conclusions and discuss future work.

Chapter 2
Background

2.1 Tools Overview

This work is based on four large software projects: abc, Soot, Eclipse and AJDT. All analyses are built under the Soot framework in the back-end of abc, and the visualization is built on top of AJDT under the Eclipse platform. In this section, we give an overview of each and a description of how they are used in this work.

2.1.1 Soot

Soot[1] [VRHS+99, VR00] is a Java program analysis and optimization framework developed by the Sable Research Group at McGill University over the past decade.

Soot supports four intermediate representations (**`IR`**s) for representing Java bytecode: Baf, Jimple, Shimple and Grimp. Although all these IRs can be used to perform analysis and transformation of Java program, Jimple is the most suitable IR. *Jimple* is a typed, stack-less, 3-address representation of bytecode. In Jimple, statements are represented by sub-types of the `Stmt` interface. There are only 15 different kinds of `Stmt` in Jimple. Compared to hundreds of different kinds of bytecode instructions, Jimple is much easier to manipulate. Stacks in bytecode are replaced by local variables in Jimple introduced by

[1] http://www.sable.mcgill.ca/soot

Soot. In addition, all local variables are given explicit names and types. These features make Jimple an ideal IR for facilitating the implementation of analyses and optimizations. Listing 2.2 shows an example of Jimple code corresponding to the Java program in Listing 2.1. Comparing to bytecode in Listing 2.3, note the compactness of instruction set and typed local variables, and that stacks are replaced by local variables prefixed with $. Soot also includes a *data-flow analysis framework* which is not IR specific. Many analyses in our project are implemented using this framework on the Jimple IR.

```
1  public int stepPoly(int x)
2  {
3     if (x < 0) {
4        System.out.println("foo");
5        return -1;
6     } else if (x <=5 ) {
7        return x * x;
8     } else {
9        return x * 5 + 16;
10    }
11 }
```

Listing 2.1 `stepPoly` in Java form [VRHS+99]

To support inter-procedural analysis, Soot constructs a *call graph*, which contains information regarding possible targets of virtual method calls, if running in the whole-program mode. The simplest call graph is constructed through Class Hierarchy Analysis (**CHA**), and both Rapid Type Analysis (**RTA**) and Variable Type Analysis (**VTA**) [SHR+00] can provide more accurate call graphs. In our project, we use call graph information to estimate targets of method calls, and since call graph constructor needs a specific entry point, the main class, of a program, **AIA** also requires this information.

There are two different points-to analyses and call graph construction frameworks in Soot: SPARK [LH03, Lho02] and Paddle [Lho06]. *Points-to analysis* is a static program analysis intended to estimate the set of locations pointed-to by a reference variable [EGH94]. SPARK is a customizable framework for inter-procedural, flow-insensitive and context-insensitive points-to analysis, and a points-to analysis is provided as part of SPARK. While gathering points-to information, SPARK also constructs a call graph on the fly utilizing the points-to information gathered, and this call graph is more precise than the

```
public int stepPoly(int)
{
   Test r0;
   int i0, $i1, $i2, $i3;
   java.io.PrintStream $r1;

   r0 := @this: StepPoly;
   i0 := @parameter0: int;
   if i0 >= 0 goto label0;

   $r1 = <java.lang.System: java.io.PrintStream out>;
   virtualinvoke $r1.<java.io.PrintStream:void
      println(java.lang.String)>("foo");
   return -1;

 label0:
   if i0 > 5 goto label1;

   $i1 = i0 * i0;
   return $i1;

 label1:
   $i2 = i0 * 5;
   $i3 = $i2 + 16;
   return $i3;
}
```

Listing 2.2 `stepPoly` in Jimple form [VRHS+99]

one generated using **CHA**. Paddle provides context-sensitive analysis, implemented using Binary Decision Diagrams (**BDD**) [BLQ+03]. In our project, points-to analysis results are used to deduce possible types of reference variables and to determine aliased variables.

Soot also has a side-effect analysis [Raz99] built-in and exposes the `SideEffect-Tester` [Lho02] interface. Side-effect analysis is a static program analysis intended to estimate variables that inspected or altered by a computation, and the *side-effect tester* tells if a variable is read/written by a computation. Two different side effect testers are implemented as part of Soot: `NaiveSideEffectTester`, which is conservative and naive, and `PASideEffectTester`, which uses call graph and points-to information. In our project, the side-effect tester is invoked to test the invariance of variables.

However, the call-graph, points-to and side effect information can only be built or gath-

```
public int stepPoly(int);
 Code:
  0: iload_1
  1: ifge 14
  4: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
  7: ldc #3; //String foo
  9: invokevirtual #4; //Method
     java/io/PrintStream.println:(Ljava/lang/String;)V
  12: iconst_m1
  13: ireturn
  14: iload_1
  15: iconst_5
  16: if_icmpgt 23
  19: iload_1
  20: iload_1
  21: imul
  22: ireturn
  23: iload_1
  24: iconst_5
  25: imul
  26: bipush 16
  28: iadd
  29: ireturn
```

Listing 2.3 `stepPoly` in disassembled bytecode form

ered if the whole program is analyzed, and **AIA** needs this information; thus **AIA** cannot analyze partial programs.

2.1.2 abc

abc[2] [ACH+05], the **aspectbench compiler** (**abc**), which is developed by a joint team from Oxford, McGill and BRICS universities, is an extensible AspectJ compiler and supports the whole of the AspectJ language. The design goals of abc are simplicity, modularity, proportionality and analysis capability. To fulfill these goals, abc is built on two established frameworks: Polyglot, a modular extension of the Java language as the frontend[3], and Soot as the backend. Abc chooses Jimple as its IR in the backend, and this provides abc the

[2] http://www.aspectbench.org

[3] In the latest version of abc, JastAddJ is added as a new and the default frontend, but the Polyglot frontend is still supported.

capability to implement sophisticated analysis. The frontend and backend are connected via the Java abstract syntax tree (**AST**) and the AspectInfo data structure. An overview of abc is given in Figure 2.1.

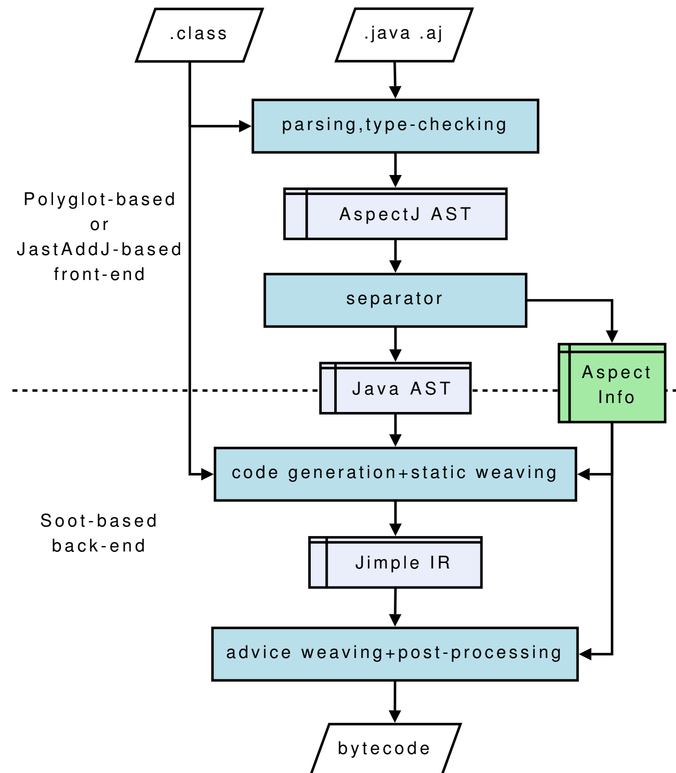

Figure 2.1 abc overall design, adapted from [dM04]

In the frontend, abc generates a pure Java AST and separates aspect information into AspectInfo. But, the Java AST may not compilable since class members injected or hierarchy modified by static crosscutting have not been reflected in it yet. Fortunately, Soot builds the Jimple IR in two stages, as shown in Figure 2.2, which describes in detail the box labeled “code generation + static weaving” in Figure 2.1. In the first stage, Soot builds merely the skeleton of the program without generating body of method, and this stage is shown as the box labeled “Soot skeleton generation”. In the second stage, Soot generates method bodies from the bytecode or the Java AST, and this stage is shown as the box labeled “Soot Jimple body generation”. Therefore, abc can perform the static crosscutting weaving between these two stages, shown as the box labeled “Skeleton weaving”. To

evaluate the impacts caused by static crosscutting, **AIA** collects the "*Pre-weave hierarchy*" information between "Soot skeleton generation" and "Soot Jimple body generation" by querying "*Jimple skeleton*".

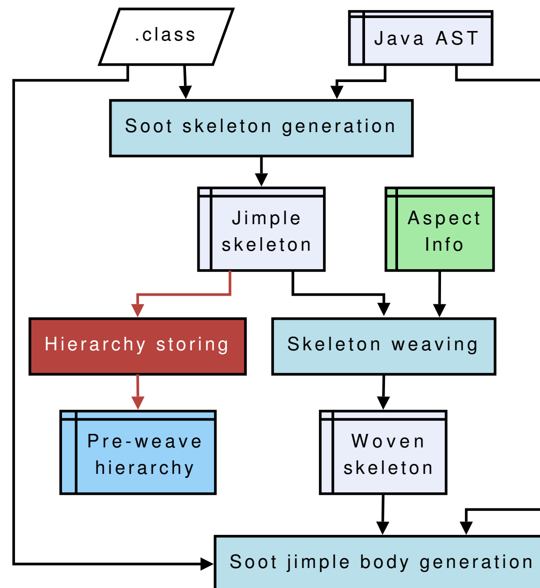

Figure 2.2 Code generation and static weaving of abc, and the pre-weave hierarchy recording phase of **AIA** in static weaving, extended from [ACH+05]

After "Advice weaving", abc allows customized "Analyses and optimization" on "*Woven Jimple*", see Figure 2.3, which describes in detail the box labeled "advice weaving + post-processing" in Figure 2.1. The advice weaver in abc is structured as a "Shadow finder", followed by a "Matcher", followed by a "Weaver", which results in "Woven Jimple". The post-processing that comes after the "Woven Jimple" consists of "Analysers" and "Optimizers". All analyses in **AIA** are implemented in the post-processing phase as an analyser — "*Impact analyser*". Analyses regarding impacts caused by static crosscutting also query the "*Pre-weave hierarchy*" information recorded in the static weaving phase. Figure 2.4 shows this in detail. The analysis phase comes after the woven Jimple has been created, and this is where "Impact analyser" fits in. This is an ideal location for "Impact analyser" because at this point all static and advice weaving have been done, and all advice bodies have been translated into normal Java methods (represented in Jimple). In addition, we also have both all of the information about the original aspects stored in `AspectInfo`

and “Pre-weave hierarchy” information recorded in the static weaving phase. As indicated by Figure 2.4, “Impact analyser” is one of analysers applied at this stage of abc. Some other abc analysers are used to optimize the weaving, and in those cases the code may be rewoven. However, “Impact analyser” only needs to run on the first pass of the weaver; it executes the analyses and then produces the “*Impact report*”.

In addition to the reason mentioned in the end of Section 2.1.1, as most of the work in **AIA** is done after weaving, our project does not work on programs with compile-time errors.

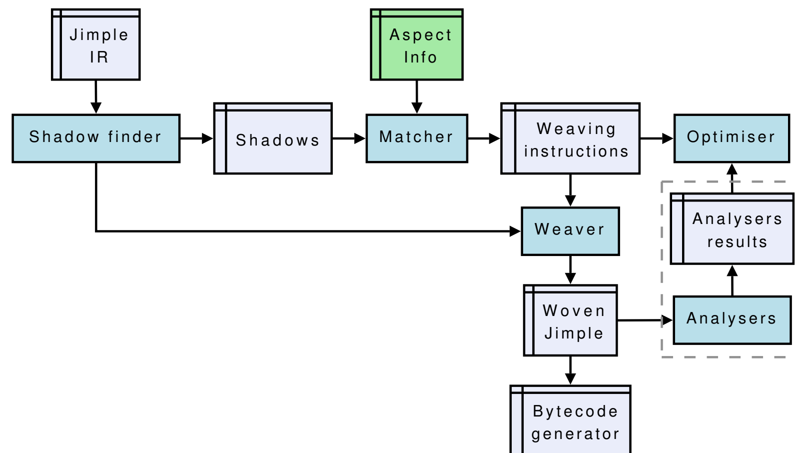

Figure 2.3 Advice weaving and post-processing of abc [dM04]

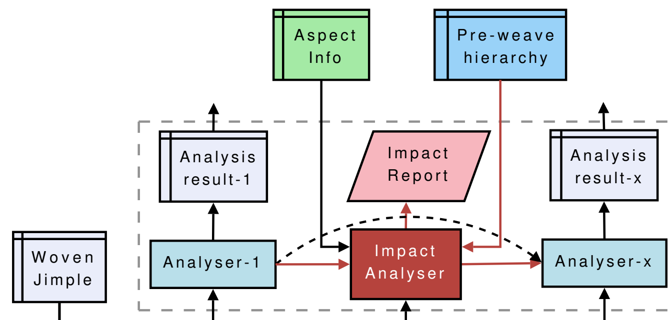

Figure 2.4 AIA in post-processing

2.1.3 Eclipse

Eclipse[4] [Inc03] is an open-source, extensible integrated development environment (**IDE**). The basis for Eclipse is the Rich Client Platform (**RCP**), including OSGi, Core, the Standard Widget Toolkit (**SWT**), JFace and the Eclipse Workbench.

SWT provides the foundation for the entire Eclipse user interface (**UI**) and implements Eclipse's widgets. Unlike the Java standard Abstract Window Toolkit (**AWT**) or Swing, SWT uses native widgets whenever possible, thus producing applications that adhere very closely to the look and feel of different operating systems. Eclipse's user interface also leverages an intermediate GUI layer called **JFace**, which simplifies the construction of applications based on SWT. Plug-ins are employed in Eclipse in order to provide their functionality on top of RCP. This plug-in mechanism is a lightweight software componentry framework where one can easily add new functionality by developing a plug-in. The plug-in architecture supports writing any desired extension to the environment. We integrate our work into Eclipse as a *plug-in*, our visualization extends *menu*, *view*, *console* and *marker* upon SWT and JFace.

2.1.4 AJDT

The AspectJ Development Tools (**AJDT**)[5] provides Eclipse-platform-based tool support for AspectJ. The goal of AJDT is to deliver a user experience that is consistent with the Java Development Tools (**JDT**) when working with AspectJ projects and resources. AJDT extends the editor of JDT and adds new builder and views to JDT to support AspectJ. Our Eclipse plug-in just takes the advantage of *AJDT* as an IDE for AspectJ program, **AIA** made no direct modification or extension to AJDT.

[4] http://www.eclipse.org
[5] http://www.eclipse.org/ajdt

2.2 The Benchmarks

There are eight benchmarks in total. Six of them are selected from the Sable AspectJ benchmarks[6] and the Eclipse AspectJ examples[7]. The other two are acquired from Internet as they demonstrate interesting use of AspectJ. The benchmarks represent a wide array of applications of AspectJ, and most of them have been used by other researchers as benchmarks. Below is a list of each benchmark with a brief description of its key features.

aopbank[8]: is a simple AspectJ program demonstrating the basic use of AspectJ. It contains three aspects deal with auditing, fee charging and securing of the simple bank implementation.

bean: examines an aspect that makes Point objects into Java beans with bound properties. Point is a simple class representing points with rectangular coordinates. An aspect is declared to make Point a serializable class and a bean, and make its get and set methods support the bound property protocol.

DCM: is a checker for Java programs. It checks the Law of Demeter. DCM includes a class form checker and an object form checker, and gives AspectJ code for each of them.

exptree6[9]: is an AOP version of expression tree program. This program makes aggressive use of ”inter-type” declarations to modify the OOP structure of a program.

observer: illustrates how the Subject/Observer design pattern can be coded with aspects.

ProdLine: implements a simple yet illustrative product-line of graph algorithms and demonstrates how AOP techniques could fit in the product line context.

Tetris: is an AOP version of the Tetris game. Various aspects are declared for GUI, logic and development.

[6]http://www.sable.mcgill.ca/benchmarks/
[7]http://www.eclipse.org/aspectj/doc/released/progguide/examples.html
[8]http://www.cs.hofstra.edu/ cscccl/csc123/aop/aopbank.java
[9]http://www.cs.hofstra.edu/ cscccl/csc123/aop/exptree6.java

tracing: prints trace messages into a stream before and after constructors and methods are executed. It demonstrates the use of aspect inheritance. It firstly defines one abstract aspect containing an abstract pointcut for injecting the tracing functionality into any application classes and then implements a tracing with a concrete pointcut inside a concrete aspect. The aspects are pure observers, and this means they do not interfere the rest of the program.

Table 2.1 shows basic metrics about these benchmarks. The column SLOC shows the source lines of code (**SLOC**) of these benchmarks. Since we could not find a SLOC counter that can count the logical SLOC of AspectJ programs, we count the physical SLOC using LocMetrics[10]. The following four columns lists the number of classes, the number of aspects, the number of advice declarations, and the number of inter-type declarations (**ITD**s) respectively. These numbers are counted manually since we could not find a tool that can count these automatically.

benchmark	SLOC	classes	aspects	advice	ITDs
aopbank	177	3	3	5	1
bean	276	2	1	2	7
DCM	3435	30	4	8	2
exptree6	327	7	5	4	27
observer	268	6	2	1	11
ProdLine	1345	9	11	15	79
Tetris	1484	9	8	21	0
tracing	430	4	2	4	0

Table 2.1 Basic benchmark metrics

[10]http://www.locmetrics.com/

Chapter 3
Four Kinds of Impact

3.1 Overview

An aspect interacts with the base program mainly through advice and inter-type declarations (also known as static crosscutting), so in **AIA**, we identify impacts caused by both. In addition, we believe the behavior of a program is determined by both the state of the program and the computation done by the program. In the domain of object-oriented programming, the state of an object-oriented program is expressed as fields of classes; the computation is expressed as methods. Therefore, we identify impacts caused by advice and inter-type declarations acting on both the state and computation of the *base program*. Thus, we have four kinds of impact:

State Impact: the impact caused by advice on fields by changing values of fields.

Computation Impact: the impact caused by advice on methods by augmenting, eliminating or substituting the computation done by methods.

Shadowing Impact: the impact caused by inter-type declarations on fields causing field references to change and thus causing field shadowing[1].

[1] The problem of field shadowing has been noticed by OOP programmers and one such online discussion was written by Bras in [Bra03]

Lookup Impact: the impact caused by inter-type declarations on methods causing method lookup changes.

We define *base program* following the principle of "everything except me". For example, if we discuss the impact caused by an advice, the base program refers to the whole program except this advice; if we discuss the impact caused by an inter-type method declaration, the base program refers to the whole program except this inter-type method declaration. Therefore, based on this rule, **AIA** takes the interference of aspects into account naturally.

In Section 3.2, we first provide two example applications which we refer to throughout the thesis. We then describe these four kinds of impact in detail in the following sections. In the last section, we discuss aspect interference in detail.

3.2 Examples

Before starting our classification discussion, we first present two example AspectJ programs: *bank* and *source code repository*. We will refer these two examples when explaining our definition of impacts, presenting our analyses and talking about our experience.

3.2.1 Bank

In the *bank* example, we are simulating a system developed in an aspect-oriented way. The system is not perfectly well designed, but this just reflects the real world. Moreover, for presentation purposes, we omit certain details when implementing the system. Basically, in this example, we assume two lines of products are provided by this bank: accounts and credit cards, which are implemented in two different packages.

The package `bank.account` holds account products. In this package, there is an abstract `AbstractAccount` class (Listing 3.1) defining basic functions on accounts such as debiting, crediting, transferring funds and charging fees. The `CheckingAccount` class (Listing 3.2) simply extends the `AbstractAccount` class. The `StudentChecking-Account` class (Listing 3.3) extends `CheckingAccount` and provides fifty percent of fee discount.

```java
public abstract class AbstractAccount {

   protected Date lastVisit;
   protected int balance;
   public final int FEE = 2;

   public AbstractAccount(int money) {this.balance = money;}

   public void withdraw(int am) {balance = balance - am;}

   public void deposit(int am) {balance = balance + am;}

   public void fee() {balance = balance - FEE;}

   public void transfer(AbstractAccount other, int am) {
      other.deposit(am);
      this.withdraw(am);
   }

   public int getBalance() {return balance;}
}
```

Listing 3.1 AbstractAccount.java

```java
public class CheckingAccount extends AbstractAccount {
   public CheckingAccount(int money) {super(money);}
}
```

Listing 3.2 CheckingAccount.java

```java
public class StudentCheckingAccount extends CheckingAccount {
   public StudentCheckingAccount(int money) {super(money);}
   public void fee() {balance = balance - (int)Math.round(FEE/2);}
}
```

Listing 3.3 StudentCheckingAccount.java

```
4  public aspect AccountAspect {
5
6     before(AbstractAccount account) :
7        (execution(public void AbstractAccount+.withdraw(int))
8           || execution(public void AbstractAccount+.deposit(int)))
9           && target(account) {
10       account.lastVisit = new Date();
11    }
12
13    after (AbstractAccount account) :
14       execution (public void AbstractAccount+.withdraw(int))
15          && target(account) {
16       account.fee();
17    }
18
19    void around() :
20       execution (public void AbstractAccount+
21             .transfer(AbstractAccount, int)) {
22       System.out.println("Transfer starts at "+new Date());
23       proceed();
24       System.out.println("Transfer completes at "+new Date());
25    }
26 }
```

Listing 3.4 AccountAspect.aj

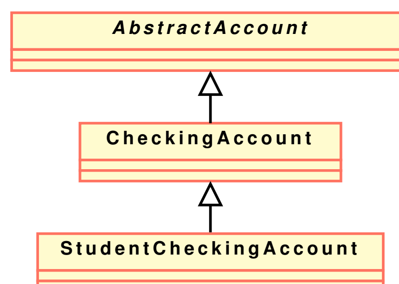

Figure 3.1 Class hierarchy of account classes

```
4   public abstract class AbstractCreditCard {
5
6       protected int balance;
7       public String bonus = "unemployment insurance";
8
9       public void payment(AbstractAccount account) {
10          int payMoney = balance;
11          account.withdraw(payMoney);
12          balance = 0;
13          System.out.println("Pay " + payMoney);
14      }
15
16      public void debit(int am) {this.balance += am;}
17  }
```

Listing 3.5 AbstractCreditCard.java

```
4   public class RewardCard extends AbstractCreditCard {
5
6       public String bonus = "1% cash back";
7
8       public void payment(AbstractAccount account) {
9           int payMoney = (int)Math.round(balance*0.99);
10          int rewardMoney = balance - payMoney;
11          account.withdraw(payMoney);
12          balance = 0;
13          System.out.println("Pay " + payMoney + ", reward " + rewardMoney);
14      }
15  }
```

Listing 3.6 RewardCard.java

In the aspect called `AccountAspect` (Listing 3.4), three advice declarations are given. The **`before`** advice records the current time as the last visit time. The **`after`** advice charges a fee after each debit. The **`around`** advice measures the time taken to make a transfer. Figure 3.1 shows the hierarchy of accounts.

```
3   public class ValueCard extends AbstractCreditCard {
4       //...
5   }
```

Listing 3.7 ValueCard.java

```
3 public class GoldCard extends ValueCard {
4     //...
5 }
```

Listing 3.8 GoldCard.java

```
4  public aspect CreditCardAspect {
5      public String GoldCard.bonus = "car rental insurance";
6
7      declare parents: ValueCard extends RewardCard;
8
9      public void GoldCard.payment(StudentCheckingAccount account) {
10         System.out.println("Can not pay with student checking account");
11     }
12 }
```

Listing 3.9 CreditCardAspect.aj

Another package `bank.creditcard` holds credit card products. In this package, there is an abstract `AbstractCreditCard` class (Listing 3.5) defining basic functions on credit cards such as debiting and payment; in addition, we assume that a credit card having the "unemployment insurance" bonus by default. The `ValueCard` class (Listing 3.7) simply extends the `AbstractCreditCard` class. The `GoldCard` class (Listing 3.8) extends the `ValueCard` class. We omit details regarding the characters of these two cards. The `RewardCard` class (Listing 3.6) extends `AbstractCreditCard` and provides "one percent cash back" reward.

In the aspect called `CreditCardAspect` (Listing 3.9), three inter-type declarations are given to simulate an upgrade to the system. The inter-type field declaration injects the `bonus` field to `GoldCard` to provide a "car rental insurance" bonus. The inter-type parents declaration makes `ValueCard` extend `RewardCard` to provide a cash back bonus to a value credit card. The inter-type method declaration injects the `payment(StudentCheckingAccount)` method to `GoldCard` to prevent a gold credit card holder from paying with a student checking account. Figure 3.2 shows the hierarchy of credit cards after applying `CreditCardAspect`.

The `Bank` class (Listing 3.10) only contains a main method, in which two `StudentCheckingAccount` objects are instantiated and three different transactions are performed.

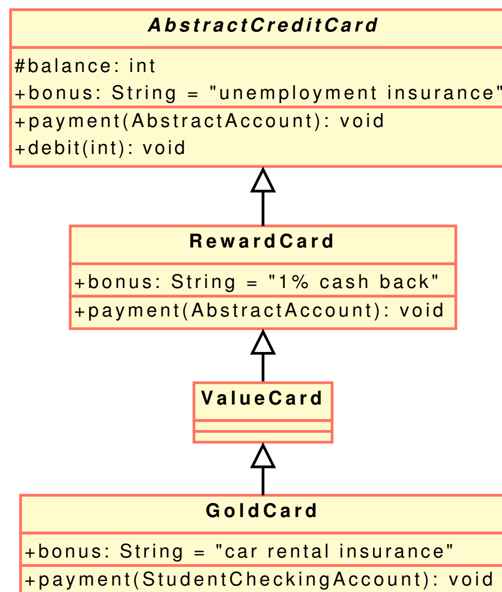

Figure 3.2 Class hierarchy of credit card classes after applying `CreditCardAspect`

After that, a `GoldCard` is instantiated, and the bonus of this credit card is queried, but queried in three different ways. Then, another `GoldCard` and `StudentCheckingAccount` are instantiated; two debits and two payments are performed, and the two payments are from the same student checking account, but in the second payment the account is cast to its run-time type — `StudentCheckingAccount`.

The output of running this program is listed in Listing 3.11. Notice that in lines 5-7, the bonus of the same gold credit card is different when querying through different types, and also the payment output in lines 8-9 is different although from the same account to the same credit card. We will explain the reasons later when we discuss *shadowing impact* and *lookup impact* in Sections 3.5 and 3.6 respectively.

3.2.2 Source Code Repository

The *source code repository* example is intended to demonstrate a legacy object-oriented system patched by aspects. In the original program, in class `SourceCodeRepository` (Listing 3.12), it requires a check of username and password every time before a retrieve or store operation (We omit the implementation of getting user input and fetching user

```java
public class Bank {

   public static void main(String [] args)
   {
      AbstractAccount acct1 = new StudentCheckingAccount(5000);
      AbstractAccount acct2 = new StudentCheckingAccount(2000);

      acct1.deposit(300);
      acct1.withdraw(200);
      acct2.transfer(acct1, 200);
      System.out.println("Account 1 balance: " + acct1.getBalance());
      System.out.println("Account 2 balance: " + acct2.getBalance());

      GoldCard goldCard1 = new GoldCard();
      System.out.println("Gold Credit Card bonus: " + goldCard1.bonus);
      System.out.println("Gold Credit Card bonus: " +
          ((ValueCard)goldCard1).bonus);
      System.out.println("Gold Credit Card bonus: " +
          ((AbstractCreditCard)goldCard1).bonus);

      GoldCard goldCard2 = new GoldCard();
      AbstractAccount stuAcct1 = new StudentCheckingAccount(1000);
      goldCard2.debit(100);
      goldCard2.payment(stuAcct1);
      goldCard2.debit(100);
      goldCard2.payment((StudentCheckingAccount)acct2);
   }
}
```

Listing 3.10 Bank.java

```
Transfer starts at Thu May 08 16:26:13 EDT 2008
Transfer completes at Thu May 08 16:26:13 EDT 2008
Account 1 balance: 5299
Account 2 balance: 1799
Gold Credit Card bonus: car rental insurance
Gold Credit Card bonus: 1% cash back
Gold Credit Card bonus: unemployment insurance
Pay 99, reward 1
Can not pay with student checking account
```

Listing 3.11 Output of bank example

```
3  public class SourceCodeRepository {
4
5     private String sourcecode;
6
7     public SourceCodeRepository() {}
8
9     private boolean login() {
10       String userIn = "mcgill", passIn = "sable"; //user input
11       String userDb = "mcgill", passDb = "sable"; //database
12       if (userIn.equals(userDb) && passIn.equals(passDb)) return true;
13       else return false;
14    }
15
16    public String getSrc() {
17       if (login()) return sourcecode;
18       else return null;
19    }
20
21    public void putSrc(String src) {
22       if (login()) { this.sourcecode = src; }
23    }
24
25    public static void main(String [] args) {
26       SourceCodeRepository repo = new SourceCodeRepository();
27       repo.putSrc("foo");
28       System.out.println(repo.getSrc());
29       repo.putSrc("junk foo");
30       System.out.println(repo.getSrc());
31    }
32 }
```

Listing 3.12 SourceCodeRepository.java

name and password from database instead assign these variables directly). However, let us assume that the repository becomes open-source, and no authentication is needed for accessing code, but a filter is needed to filter out junk programs. Therefore, in the aspect `SourceCodeRepositoryAspect` (Listing 3.13), two advice declarations are defined. The first **`around`** advice captures the `login` method and always returns `true` to bypass the login step, and the second `around` advice captures the `putSrc` method and rejects the source code if the filter (in this case just implemented as a length check) returns false.

```
3  public aspect SourceCodeRepositoryAspect {
4
5     boolean around() :
6        execution (boolean SourceCodeRepository.login()) {
7        return true;
8     }
9
10    void around(String src) :
11       execution (void SourceCodeRepository.putSrc(String))
12          && args(src) {
13       if (src.length() < 5) {
14          proceed(src);
15       } else {
16          System.out.println("Out of limit");
17       }
18    }
19 }
```

Listing 3.13 SourceCodeRepositoryAspect.aj

3.3 State Impact

AspectJ is built on top of Java, which is object-oriented. In both object-oriented programming and aspect-oriented programming, the program state is mainly defined by values of fields in classes. Therefore, our state impact focuses on how aspects can modify fields in a *base program*. Because aspects interact with the program at the granularity of advice, we define our state impact in the granularity of advice. Thus, based on our principle "everything except me", here, *base program* means the whole program except the aspect containing the advice being considered, thus, fields in base program means all fields except fields defined in the containing aspect of the advice which causes state impact. Since an advice can change fields of base classes both directly and indirectly, we classify state impact further into:

Direct state impact: is an impact caused by advice modifying fields of base classes directly in the form such as *base.field = NewValue*. In the bank example, `account.lastVisit = new Date();` (Listing 3.4, line 10) in the **`before`** advice causes a direct state impact because it writes the field `lastVisit` in the class `AbstractAccount` or its subclass.

Indirect state impact: is an impact caused by advice invoking a method or calling a chain of methods which modify fields of base classes. In the bank example, `account.fee();` (Listing 3.4, line 16) in the **`after`** advice causes an indirect state impact. If we check the method body of `AbstractAccount.fee()`, we can see that the `money` field is modified, but the `money` field may belong to `AbstractAccount` or its subclass. However, the `fee()` method is overridden in `StudentCheckingAccount`, the call `account.fee()` may dispatch to this method, but `StudentChecking-Account.fee()` also modifies `money` field. Therefore, we can state for sure that a state impact is caused by the **`after`** advice. Later, in Section 4.3.1, we will see that points-to analysis can give the precise estimation of the actual method being called and the actual class that money belongs to.

Therefore, we define *state impact* as program state change caused by advice modifying the value of fields of base classes directly or indirectly.

3.4 Computation Impact

Applying advice to a program usually changes the computation performed. An advice will match a collection of shadows in the base program. Here, following our "everything except me" principle, *base program* means the whole program except the advice being considered. **`Before`** and **`after`** advice can add computation before or after the shadow; however, **`around`** advice can have impact on whether or not the shadow code executes, depending on how the body of the advice calls **`proceed`**. Thus, in order to handle **`around`** advice, we first introduce the concept of *exact-proceed*.

3.4.1 Exact-proceed

We define an *exact-proceed* as a proceed call that fulfills the following three conditions:[2]

same arguments: the same argument values as found in the join point must be passed by the **`proceed`** call;

[2]Similar conditions have been defined by Recebli [Rec05] and Rinard. et. al. [RSB04].

same return value: the value returned by **`proceed`** must be returned by the advice without modification; and

no abrupt exception: no exception stops the reachability of **`proceed`**.

The idea behind these conditions is that the *same arguments* and *same return value* conditions ensure that the original computation at the join point is executed and the same value is returned, whereas the *no abrupt exception* condition ensures that the computation is always executed as it was in the base program.

We also use the concept of *live* and *dead* advice. For a specific program, we say that an advice is *live* if it matches at least one shadow and it is *dead* if it does not match any shadows.

Given these definitions we now define *invariant advice* and four flavors of *variant advice*.

3.4.2 Invariant Advice

We define an advice to be invariant if it adds no new computation to any shadow, nor removes any computation from any shadow. For **`before`** and **`after`** advice either: (a) the advice is *dead*, *i.e.*, it does not match any shadow, or (b) the advice body is empty. For **`around`** advice, either: (a) the advice is dead, or (b) the body of the advice is composed of exactly one exact-proceed. Invariant advice is not very interesting, and if we find invariant advice it is likely to indicate a bug in the program (for example, some AspectJ compilers (*e.g.*, abc, ajc) give a warning when an advice does not match anywhere in the program).

3.4.3 Variant Advice

Variant advice is much more interesting and useful than invariant advice. The idea is that we want to know if the advice adds computation to matching shadows, eliminates code at shadows, or replaces code at shadows. We classify variant advice accordingly into the following kinds of computation impact:

Addition: After applying the advice (weaving), the matched shadows in the base program always execute unchanged, and new computation is added. Logging advice [Lad03]

		`before` `after`	`around`
Invariant		*dead* or empty body	*dead* or exactly one *exact-proceed* with no additional computation
Variant	**Addition**	*live* and non-empty body	*live* and at least one *exact-proceed* on every path, plus additional computation
	Elimination		*live* and empty body
	Definite-Substitution		*live* and no **`proceed`** on any path
	Conditional-Substitution		*live* and at least one *exact-proceed* on one or more paths but no **`proceed`** on other paths
	Mixed		*live* and have no computation impact above

Table 3.1 Classification of computation impacts

is a typical example. In the bank example, all three advice definitions have addition computation impacts.

Elimination: After applying the advice, the matched computation in the base program is removed, and no new computation is added. Advice hiding method functionality often cause elimination impacts. In the source code repository example, the **`boolean around`** advice (Listing 3.13, line 5) that short-circuits authentication has elimination computation impact.

Definite-Substitution: After applying the advice, the matched computation in the base program does not execute at all, and new computation is added. In this case the advice replaces a functionality in the base program with a brand-new one. An example is an advice replacing an old algorithm with optimized algorithm.

Conditional-Substitution: After applying the advice, the matched computation in the base program may or may not execute depending on some conditions, and new computation is always added. In this case an advice either replaces a functionality in the base program or adds new computation with the matched shadows executing unchanged, depending on the condition. An example is an advice that introduces a condition check to determine if the old algorithm should be replaced by a different algorithm. In the source code repository example, the **`void around`** advice (List-

ing 3.13, line 10) that filters source code has conditional-substitution computation impact.

Mixed: In this case, after applying the advice, the effect on the matched computation in the base program can't be determined statically. At run-time, it may or may not execute the matched computation when following different paths, or even when following the same control flow path. For example, an advice containing a **`proceed`** call one of whose arguments is user input causes mixed computation impact.

In Table 3.1, we present a summary of our categorization expressed in terms of our definitions of live/dead advice and the definition of exact-proceed. Note that **`before`** and **`after`** advice is either invariant or has addition impact, because the original shadow code is never removed. However, **`around`** advice can have different impacts, depending on how **`proceed`** is used in the body of the advice.

3.5 Shadowing Impact

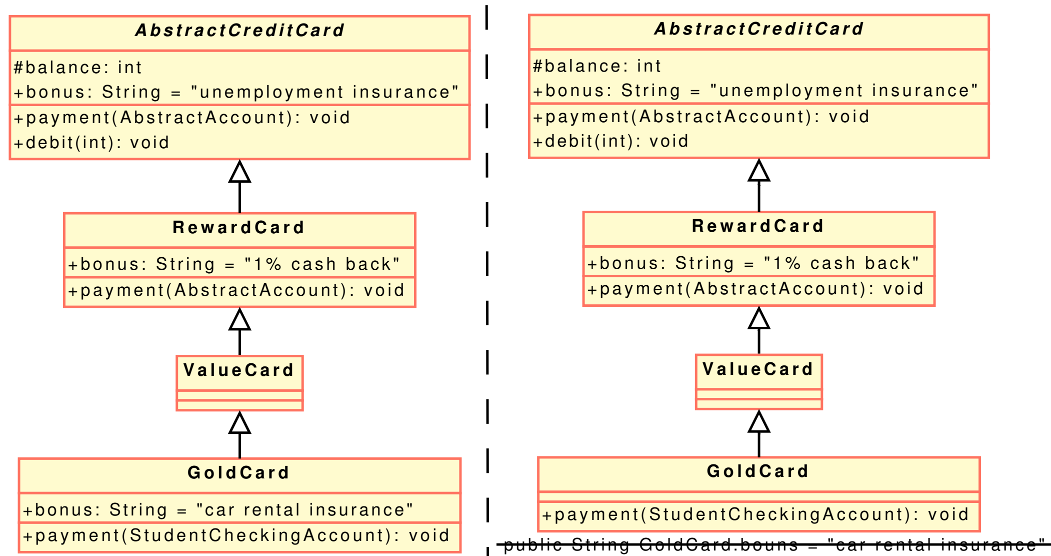

Figure 3.3 Comparison of class hierarchy of credit card classes with and without `GoldCard.bonus` inter-type declaration

Inter-type field declarations can inject new fields into classes. These field declarations will neither change state nor computation of a base program. However, a new declared field may cause name shadowing if the name of the new field is the same as the name of a field that is declared in the inter-type target class' super classes and is inheritable to the target class, and further causes the reference of the field on a variable with the type of the inter-type target class changes before and after applying the inter-type declaration. A similar situation called "variable shadowing" exists in the context of OOP, thus we name this impact as *shadowing impact* and define it as the change of field reference in the base program after applying an inter-type declaration, and following our principle, the *base program* means the whole program except the inter-type declaration being considered. Essentially, a shadowing impact is caused by name shadowing, but the obliviousness of AOP increases the possibility of an occurrence of name shadowing, and we believe understanding the occurrence of it helps reasoning about an aspect-oriented program and reduces resulting bugs.

In the bank example, the inter-type field declaration `GoldCard.bonus` in the aspect `CreditCardAspect` (Listing 3.9, line 5) causes shadowing impact because after applying the field declaration, `[GoldCard].bonus`[3] refers to the new declared `bonus` field in `GoldCard` instead of the `bonus` field in `RewardCard`. We state the field in `RewardCard` instead of `AbstractCreditCard` because the base program includes the inter-type parent declaration `ValueCard extends RewardCard` (line 7 in Listing 3.9). This is clear if comparing the class hierarchy after weaving the whole `CreditCardAspect` and the class hierarchy without weaving the `GoldCard.bonus` declaration, as shown in Figure 3.3.

In addition to inter-type field declarations, inter-type parent declarations also can cause a shadowing impact. By introducing a new parent to a class, the class may inherit a new field which has the same name as the name of a field in the class' old ancestors, thus reference to the field changes to the new inherited field. The inter-type parent declaration in the aspect `CreditCardAspect` (Listing 3.9, line 7) causes a shadowing impact on `ValueCard`. `[ValueCard].bonus` refers to the `bonus` field currently inherited from `RewardCard` instead of the `bonus` field in `AbstractCreditCard`, which originally is the parent of `ValueCard`, as shown in Figure 3.4. However, this parent declaration causes

[3] In this thesis, we denote the access of field `f` of a variable of type `Type` as `[Type].f`.

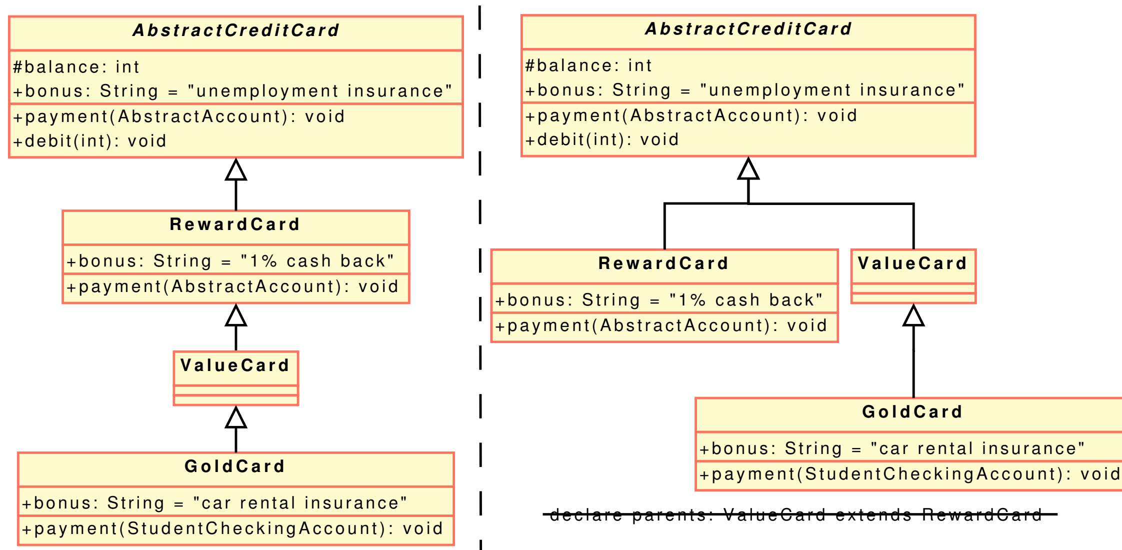

Figure 3.4 Comparison of class hierarchy of credit card classes with and without `ValueCard extends RewardCard` inter-type declaration

no impact on `GoldCard` since `GoldCard` has its own declaration of `bonus` field, which is injected by inter-type field declaration at line 5 of Listing 3.9. Remember that here *base program* means the whole program except the inter-type declaration being considered, thus the effect of `GoldCard.bonus` declaration is considered to be part of the base program.

3.6 Lookup Impact

An inter-type declaration can also inject new methods into classes. This kind of method declaration will neither change the state or computation of the base program. However, the method lookup may be changed due to a new declared method, thus the program may change its behavior or even become broken. Therefore, we introduce *lookup impact* and define it as the changing of lookup of a method invocation in the base program before and after applying an inter-type declaration, and the *base program* refers to the whole program except the involved inter-type declaration by adhering to our "everything except me" principle.

An inter-type method declaration can cause lookup impact. In the bank example, the

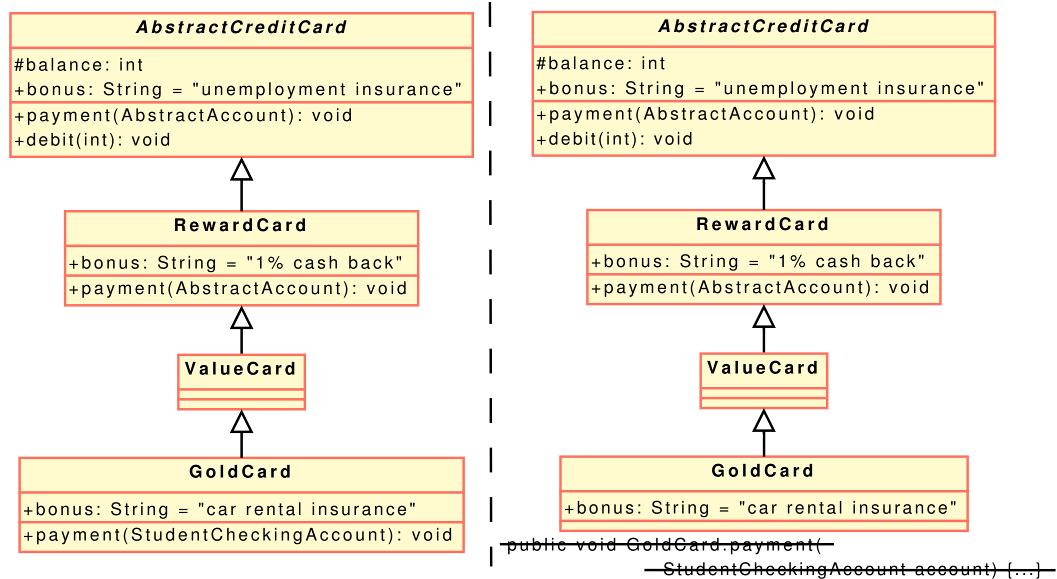

Figure 3.5 Comparison of class hierarchy of credit card classes with and without `GoldCard.payment()` inter-type declaration

inter-type method declaration `GoldCard.payment(...)` at line 9-11 in `CreditCardAspect` (Listing 3.9) causes lookup impact. If we remove these lines, the class hierarchy is shown in the right side of Figure 3.5, the call to `payment(StudentChecking Account)` on a receiver of type `GoldCard` (denoted as `[GoldCard].payment(StudentCheckingAccount)`[4]) will match to `RewardCard.payment(AbstractAccount)`. However, in the left side of Figure 3.5, we can see after applying the `GoldCard.payment()` inter-type method declaration, the same call will match to `GoldCard.payment(StudentCheckingAccount)`. Therefore, this inter-type method declaration causes the change of lookup of the method invocation `[GoldCard].payment(StudentCheckingAccount)`, thus causes a lookup impact.

An inter-type parent declaration can also cause a lookup impact. In the bank example, the inter-type parent declaration `ValueCard extends RewardCard` at line 7 in `CreditCardAspect` (Listing 3.9) causes a lookup impact that affects the lookup of `[ValueCard].payment(AbstractAccount)` and `[GoldCard].payment(AbstractAccount)`. Referring to Figure 3.4, we can see `[ValueCard].payment(AbstractAccount`-

[4] In this thesis, we denote a method invocation `foo()` on a receiver of type `Type` as `[Type].foo()`.

`count)` and `[GoldCard].payment(AbstractAccount)` matches to `AbstractCred-itCard.payment(AbstractAccount)` before applying this inter-type parent declaration, and matches to `RewardCard.payment(AbstractAccount)` instead, after applying this inter-type parent declaration.

3.7 Aspect Interference

In Section 3.1, we mentioned that our analysis covers aspect interference naturally as a result of our definition of base program. In this section, we will discuss other possible definitions of base program and how aspect interference is taken into account for the above four kinds of impact under our definition.

3.7.1 Java part is the base program

A reasonable definition of base program might to consider the base program as the program before any aspects woven into it. Indeed, we have considered this definition, but we discovered that it is hard to automatically identifying the base program from a complex system, and it causes trouble when analyzing aspect interference. The program may not compile at all if we only consider the AspectJ element being analyzed and try to weave this element only into the Java part of the program. Further, even if the program compiles, the aspect interference must be discovered iteratively, or it might not be discovered at all if ordering is important amongst aspects elements applied to the base program. [Kat06]

3.7.2 Everything except me

However, under our "everything except me" principle, aspect interference is covered naturally.

For state impact, we define the *base program* as the whole program except the aspect declaring the advice causing state impact. Therefore, if an advice changes the state of other aspects, it will be captured by **AIA**.

For computation impact, we define the *base program* as the whole program except the advice being considered. Hence, if an advice matches shadows in other aspects, even within

other advice bodies in the same aspect, **AIA** will cover the impact on these situation too.

For shadowing impact and lookup impact, we define the *base program* as the whole program except the inter-type declaration being considered. Thus, if other inter-type declarations also work on the target of the inter-type declaration being considered, **AIA** will analyze impact caused by the inter-type declaration being considered after other inter-type declarations are applied. The shadowing impact caused by `GoldCard.bonus` and `ValueCard extends RewardCard` in `CreditCardAspect` (Listing 3.9) in bank example discussed in Section 3.5 is an example.

Chapter 4
State Impact

In Section 3.3, we introduced *state impact* and defined it as program state change caused by advice modifying the value of fields of base classes in the *base program*, which refers to the whole program except the aspect containing the advice being considered. Moreover, we classified state impact further into *direct state impact* and *indirect state impact* based on if the field value modification is direct or indirect. In this chapter, we give a precise definition of state impact and describe the static analysis for approximating state impact.

4.1 Definition

In general, *state impact* is caused by writing to fields.

A *direct state impact* is caused by statements located inside the advice body writing to fields of base classes directly, and the form of the statement is like `base.id=rhs`. The consequence of this statement is that the field "`id`" of the class `baseclass`, which is the class corresponding to the run-time type of `base`, is changed by the advice. In addition, `base.id` may be a static reference, in which case `baseclass` is the class `base` itself. After knowing what happened, the report should look like: *advice statement* changes *id* of *baseclass*.

An *indirect state impact* is caused by a method call or a chain of method calls in the advice body where the invoked methods contain statements directly writing to fields in base classes in the form of `base.id=rhs`. However, since the method may (very likely) contain

more than one direct field writing statement, to get an overall view of the modification made by the advice statement, the field modification information is better to be aggregated and grouped by field. In addition, to get information about how exactly these modifications happened, these direct modification statements lying in invoked methods should also be reported, like *evidences*. Therefore, the report would look like: *advice statement* changes: *id_1* of *baseclass_1,...,baseclass_n*, ..., *id_n* of *baseclass_1,...,baseclass_m*; evidences: *method statement x* changes *id_x* of *baseclass_x*, ..., *method statement y* changes *id_y* of *baseclass_y*.

4.2 Analysis

Based on the definition of state impacts, our state impact analysis is designed to discover all field write accesses caused by advice, directly or indirectly. At the conceptual level, the algorithm is presented in Algorithm 4.1 and Procedure `Analyze(`*body, adstmt*`)` on page 37.

Algorithm 4.1: State impact analysis

```
1 foreach advice ad in the application do
2     get Jimple body adbody of ad;
3     Analyze(adbody, null);
4 end
```

Our analysis works at the granularity of advice; we first acquire all advice information from abc after the first pass of weaving. Then, we iterate over each advice to perform the actual state impact analysis. As explained in Section 2.1.2, after the first pass of weaving, all advice bodies have been transformed to standard Java methods represented in Jimple IR. Thus, we can utilize the Soot analysis framework to perform our analysis.

4.2.1 Direct State Impact

To discover direct field modifications, we need to check if the definition of an assignment statement is referring to a field in base classes. Since Jimple is a three-address IR, we actually only need to check if the left hand side of an assignment statement is

Procedure `Analyze`(*body, adstmt*) in Algorithm 4.1

```
1  initialize cacheSet to empty;
2  foreach statement stmt in body do
       /* deal with field direct modification                    */
3      if lhs of stmt is FieldRef in the form of base.field then
4          if base.field instanceof StaticFieldRef then
5              if adstmt == null then              /* in advice body */
6                  record direct state impact on [field] in [base];
7              else
8                  record evidence on [field] in [base] of adstmt;
9              end
10         end
11         if base.field instanceof InstanceFieldRef then
12             get points-to set ptset of base;
13             get classes cset in ptset;
14             if adstmt == null then              /* in advice body */
15                 record direct state impact on [field] in [cset];
16             else
17                 record evidence on [field] in [cset] of adstmt;
18             end
19         end
20     end
       /* deal with modifications by invoking method             */
21     if stmt contains method invocation then
22         foreach target method by querying call graph do
23             if stack !contains method and cacheSet !contains method then
24                 get Jimple body mbody of method;
25                 put method into stack;
26                 put method into cacheSet;
27                 Analyze(mbody, (adstmt == null? stmt : adstmt));
28                 pop method from stack;
29             end
30         end
31         if adstmt == null then                  /* in advice body */
32             aggregate evidences of stmt;
33             record indirect impact caused by stmt;
34         end
35     end
36 end
```

referring to a field in base class. The field can be either a static field having the form `Class.field` or an instance field having the form `object.field`. In Soot, they are represented by two interfaces; the former is called `StaticFieldRef`, and the latter is called `InstanceFieldRef`. Therefore, we just need to iterate through all statements and check if their `lhss` are a `StaticFieldRef` or an `InstanceFieldRef`. Moreover, since we are trying to find all possible field modifications, no matter whether the modification happens in a branch path or not, we consider the modification as may-happening and as having direct state impact.

Beside knowing where the impact happens, we need to know what happens, i.e. we need to know which field of which class is modified. The field name can be easily acquired by querying the `FieldRef` object. For class information, in the case of a `StaticFieldRef`, we can easily get its declaring class. However, in the case of an `InstanceFieldRef`, the base object of the field may point to objects with different types in the context of static analysis; thus, we need points-to analysis to determine the set of memory locations that the base object may point to, and then we can get the set of base classes that the field may belong to. In the bank example, the points-to analysis will tell us that the actual type of the `account` in “account.lastVisit = new Date();”(Listing 3.4, line 10) in the **`before advice`** is `StudentCheckingAccount` since we instantiated a `StudentCheckingAccount` in `Bank` (Listing 3.10, line 7, 8).

All the information regarding where the impact happens and what is the impact is recorded and reported.

4.2.2 Indirect State Impact

As stated in our classification, an advice can also cause indirect state impact by calling methods which modify fields of base classes; thus, our analysis also checks indirect state impact.

If modifying fields indirectly, a statement inside an advice must call a method. The method being called may modify fields directly, or call other methods that modify fields, so we check for state impacts caused by all methods being called transitively. Moreover, call cycles, due to recursion, in the transitive call graph are resolved by maintaining a call

stack and checking if a method is in the stack before entering and analyzing it.

For all indirect field modifications analyzed during traversing methods called transitively, we recorded them as evidences to support our indirect state impact. In the same manner as with direct state impact, we record both where and what happens for each piece of evidence. Moreover, to give an overall view of the indirect state impact to programmer, before generating the report, we aggregate all points-to sets recorded while traversing the transitive call graph; we grouped them by field by calculating the union of these points-to sets.

4.2.3 Distinguish Direct and Indirect State Impact

As shown in Algorithm 4.1 and Procedure `Analyze(`*body, adstmt*`)`, our analysis detects direct and indirect state impact in one pass. When we encounter a field modification statement, we need to tell whether it causes direct state impact or is just an evidence of an indirect state impact. We distinguish this by checking if the statement is inside the body of advice being considered or not.

Basically, we check this by testing if the recursive procedure `Analyze(`*body, adstmt*`)` is called on the top level or not. The second formal *adstmt* of `Analyze` works as a flag marking if `Analyze` is at the top level and also records the original advice statement causing the chain of indirect state impact. By checking if `adstmt`(at line 5 and 14 in Procedure `Analyze(`*body, adstmt*`)`) is null, we can tell whether a field modification statement is causing direct state impact or just is an evidence of indirect state impact.

If *adstmt* is null (line 5 and 14), it means `Analyze` is on the top level, *i.e.*, `Analyze` is working on the advice body being considered, *i.e.*, the field modification statement causes direct state impact; otherwise, the statement is just an evidence of the indirect state impact caused by *adstmt*, which is the statement invoking methods at the first time. In addition, the *adstmt* is handed on at line 24 in Procedure `Analyze(`*body, adstmt*`)` to keep track of the originating statement in the advice body being considered.

4.3 Experimental Results

First, we explore our analyses with two examples: the *bank* example discussed in Section 3.2.1 and the *glass and table* example presented by Elçin [Rec05].

4.3.1 Examples

Bank

A segment of our analyses report related to advice in the bank example is shown in Listing 4.1. As expected and discussed in Section 3.3, our report shows that the **`before`** advice defined in AccountAspect.aj (Listing 3.4) at line 6-11 has a direct state impact of writing field `lastVisit` in `StudentCheckingAccount` class, and the points-to analysis gives a precise estimation of the type of `account` (line 10). Our analysis also reports that the `lastVisit` field is declared in class `AbstractAccount`.

The report also shows that the **`after`** advice at line 13-17 has an indirect state impact on the `money` field of `StudentCheckingAccount`, and this indirect state impact is caused by the statement in Account.java at line 16 column 20-39, which is the `money = money - FEE` statement in `Account.fee()`. Therefore, our analyses reports not only exactly what happens, but also how it happens, so programmers can use our analyses report to understand and analyze state impacts in very straightforward way. With the evidence report, programmers can trace back to the exact point where fields are written. Later, in the glass and table example, we will show how our state impact can help reveal bugs.

In addition, the report shows that the **`around`** advice at line 19-25 does not cause any state impact, as we expected.

Glass and Table

This program is presented by Elçin in his master thesis [Rec05]. In this example, a `Table` contains a `Glass`, and `Table.move()` calls `Glass.move()`, thus if the table moves, the glass also moves. Then, an aspect applies to the program. In this aspect, there is an **`after`** advice that matches the execution of `Glass.move()`, and this advice calls `Table.move()`

```
AccountAspect.aj:6,1-11:2 Advice: before(bank.account.AbstractAccount
    account)
state impact:
   bank\account\AccountAspect.aj:10,2-32
      direct state impact:
      field [lastVisit](declared in bank.account.AbstractAccount) in
         [bank.account.StudentCheckingAccount]
addition computation impact

AccountAspect.aj:13,1-17:2 Advice: after(bank.account.AbstractAccount
    account)
state impact:
   bank\account\AccountAspect.aj:16,2-15
      indirect state impact:
      field [balance] in [bank.account.StudentCheckingAccount]
      evidence:
      bank\account\StudentCheckingAccount.java:5,20-62 field
         [balance](declared in bank.account.AbstractAccount) in
         [bank.account.StudentCheckingAccount]
addition computation impact

AccountAspect.aj:19,1-25:2 Advice: void around()
no state impact
addition computation impact
```

Listing 4.1 Report of analyzing advice in the bank example

to accomplish the purpose of moving tables if the glass moves. If we examine both the aspect and the base program, we can discover this advice causes a call cycle, which causes an infinite loop. Running the woven program will therefore cause a `StackOverflowError` error.

Listing 4.2 shows the report of our analyses on glass and table program. We can see that our report points out the second advice has state impacts both on `Glass` and `Table`. Programmers could utilize this information to conclude this advice does something other than expected. The advice is designed to move the table when glass moves, but according to our report, it also changes the position of table, this should not happen; thus, there is something wrong. Moreover, if the programmer follows our evidence reports and checks Table.java and Glass.java, he/she will find these statements are defined in `Glass.move()` and `Table.move()`, so this advice actually transitively calls both `Glass.move()` and `Table.move()`. Therefore, the programmer should very easily conclude that there is a

call-cycle. For this simple example, switching back and forth between different source files perhaps can be handled, but in larger applications the programmer needs more direction such as provided by our impact report.

```
GlassAspect.aj:6,1-10:2 Advice: after(glass.Table t)
no state impact
addition computation impact

GlassAspect.aj:12,1-17:2 Advice: after(glass.Glass g, int dx, int dy)
state impact:
   glass\GlassAspect.aj:16,3-21
      indirect state impact:
      field [y] in [glass.Glass, glass.Table]
      field [x] in [glass.Glass, glass.Table]
      evidence:
      glass\Table.java:18,2-9 field [x](declared in glass.Table) in
         [glass.Table]
      glass\Glass.java:8,2-9 field [x](declared in glass.Glass) in
         [glass.Glass]
      glass\Table.java:19,2-9 field [y](declared in glass.Table) in
         [glass.Table]
      glass\Glass.java:9,2-9 field [y](declared in glass.Glass) in
         [glass.Glass]
addition computation impact
```

Listing 4.2 Report of analyzing the glass and table example

4.3.2 Benchmarks

Table 4.1 shows statistics of analyzing these benchmarks. The second column "C&A" shows the total number of classes and aspects. The third column shows the total number of advice declarations. The next two columns shows the number of direct and indirect state impacts respectively. The last column shows the number of classes and aspects checked when discovering these state impacts manually by examining the source code of these benchmarks. Basically, we started from the source code of all aspects and discovered all advices. For each advice, we read its source code and determined if a statement causes a state impact. If a statement contains a method call, we discovered the target methods and examined the source code of target methods. During this process, we recorded the number

of aspects and classes visited, including the classes that have to be visited to determine target methods of a method call.

benchmark	C&A	advice	direct	indirect	C&A manually
aopbank	6	5	1	1	4
bean	3	2	0	2	3
DCM	34	8	0	9	6
exptree6	12	4	0	10	9
observer	8	1	0	1	8
ProdLine	20	15	1	23	19
Tetris	17	21	0	19	14
tracing	6	4	0	0	2

Table 4.1 Basic statistics about benchmarks

From Table 4.1, we can find that for most benchmarks, when discovering these state impacts manually, at least 70% of classes and aspects need to be checked. When we discovered state impacts in DCM, we found the length of the call chain causing an indirect state impact was nine. In addition, the size of these benchmarks are not large. Therefore, we can image the difficulty of discovering and understanding state impacts in a large-scale application. However, by using our tool, it only takes few minutes to get complete state impact information.

In addition, for the tracing benchmark, we state that the aspects are pure observers when introducing it in Section 2.2. From the above table, we can find that no state impact is caused, as expected.

Chapter 5
Computation Impact

In Section 3.4, we introduced *computation impact* and defined it as the computation change of shadows in the *base program*(the whole program except the advice being considered) caused by applying advice to the matched shadows in the base program. In addition, we classify advice into *invariant advice* and *variant advice*. Invariant advice causes no computation impacts, whereas the impact of variant advice are further classified into *addition*, *elimination*, *definite-substitution*, *conditional-substitution* and *mixed* computation impacts. In this chapter, we give a precise definition of computation impact and describe the static analysis for approximating computation impact.

5.1 Definition

In order to categorize variant advice, we require static analysis to determine the behavior of proceed. In Section 3.4, we defined *exact-proceed* as `proceed` calls that fulfill the *same arguments*, *same return value* and *no abrupt exception* conditions; therefore, there should be three corresponding static analyses to test these conditions as defined below.

same arguments: The same argument values as found in the join point must be passed by the `proceed` call in the same order as they are captured in the join point. In particular, if an argument is a reference type, there is no change to its field.

same return value: The value returned by **`proceed`** must be returned by the advice without modification. In particular, if the return value is a reference type, there is no change to its field.

no abrupt exception: No checked exceptions stop the reachability of **`proceed`**. Only checked exceptions are considered because every Java statement may throw unchecked exceptions.

Given the exact-proceed information, different kinds of computation impacts can be defined as follows:

Elimination: impact is caused by an **`around`** advice having an empty body. For an around advice declaring void return type, the body should contain nothing; for an around advice declaring non-void return type, the body can contain only a return statement.

Addition: impact is caused by a **`before`** or **`after`** advice whose advice body is not empty; or an **`around`** advice that has at least one *exact-proceed* on every path, plus additional computation.

Definite-substitution: impact is caused by an **`around`** advice having a non-empty body and having no **`proceed`** on any path.

Conditional-substitution: impact is caused by an **`around`** advice having at least one *exact-proceed* on one or more paths but not on all paths and having no **`proceed`** call at all on all other paths.

Mixed: impact is caused by an **`around`** advice that does not cause any of the above impacts.

5.2 Analysis

Based on the above definition, our computation impact analysis is designed on the granularity of advice and executed after the first weaving of abc. It classifies advice into addition, elimination, definite-substitution, conditional-substitution, and mixed based on the kind of

advice and exact-proceed analysis result. At the conceptual level, our algorithm is presented in Algorithm 5.1.

Algorithm 5.1: Computation impact analysis

```
1  foreach advice ad in the application do
2      if ad is before or after and not empty then addition;
3      if ad is around then
4          if empty body then elimination;
5          else
6              if no proceed then definite-substitution;
7              else
8                  exact-proceed analysis;
9                  if at least one exact-proceed in every path then addition;
10                 else if either exact-proceed or no proceed in every path then
                   conditional-substitution;
11                 else mixed;
12             end
13         end
14     end
15 end
```

5.2.1 Exact-proceed Analysis

As discussed in Section 3.4, our computation impact classification relies on the information regarding *exact-proceed* statements. Thus, to check if a proceed is an *exact-proceed*, we implement three intra-procedural analyses to check if those three conditions, *i.e.*, *same arguments*, *same return*, and *no abrupt exception*, are satisfied using the `UnchangedParamAnalysis`, `UnchangedReturnAnalysis` and `ExceptionBeforeProceedAnalysis`, respectively. Although they are all intra-procedural analyses, the first two utilize inter-procedural analysis results, *i.e.*, points-to and side-effect analysis results.

`UnchangedParamsAnalysis`

This analysis collects all variables/expressions (represented by `Value` in Soot) that have the same value as the method's arguments. The result of this analysis is used to check the

same arguments condition. The specification of the data flow analysis is presented by the following six rules:

1. For each program point, we approximate a set of pairs, where each pair has the form `(index,v)`, representing the `v` has the same value as the `index`th argument.

2. At a program point `p`, if pair `(i,x)` is in this set then this means that the variable `x` denotes exactly what the `i`th parameter denoted at entry. In particular, if the `i`th parameter is a reference type, then `(i,x)` is in the set at program point `p`, only if `x` refers to the same object as the `i`th parameter and there has been no write to a field of the object between the entry and program point `p`.

3. It is a forward analysis.

4. The confluence operator is intersection:

$$in(n) = \bigcap_{p \in pred(n)} out(p)$$

   where: `out(p)` – means the set at the output of a node in the CFG, which is also the data set flowing out of the node; `in(n)` – means the set at the input of a node in the CFG, which is also the data set flowing into the node. The intersection operation means at a control-flow merge point only pairs belonging to both paths become part of the set after the merge.

5. The data flow equation for a node `s` is

$$out(s) = (in(s) - kill(s)) \cap gen(s)$$

   where: `out` and `in` have the same meaning with those in rule 4. The following are the `gen` and `kill` rules at a node:

   - Gen due to assign statement: if `rhs` of an assign statements in the form `lhs = rhs` is in the set, in the form of `(index,rhs)`, generate `(index,lhs)` pair for `lhs` with the same `index`.

- Kill due to assign statement: a statement of the form `lhs = rhs` kills any pair associated with `lhs`.
- Kill due to reference: for all statements of the form `a.f = b`, we kill any pair associated with `a`.
- Kill due to alias: if a pair `i, v` is being killed, any pair `i, v'` should be killed if `v` and `v'` point to the same location, *i.e.*, the `PointsToSet` of `v` and `v'` have a non-empty intersection.
- Kill due to method call: if the statement contains a method call, it may write to objects. Thus, for each pair in the set, say `(i,o)`, we check (use Soot's side-effect analysis) to see if there in an interference with the method call and any field of `o`. If there is an interference, then we kill the pair, because the state of the object may have been changed, and it may no longer denote the same value as at the entry point.

6. Starting approximations are as follows:

   - `out(start)` = $\{$`(1,arg`$_1$`), ..., (n,arg`$_n$`)`$\}$. It is safe to say each argument has the same value with itself.
   - `out(other) = universal set`, which says that every `Value` may have the same value of every argument.

The analysis is implemented under the forward data flow analysis framework in Soot. After the data-flow analysis reaches the fix point, we can query the data flow set to test the *same arguments* property of **`proceed`** calls as follows: for a statement `p` containing a **`proceed`** call — `proceed(a`$_1$`, ..., a`$_n$`)`, we fetch out the data set flowing into `p`, *i.e.*,`in(p)`, and then check if all pairs `(1,a`$_1$`), ..., (n,a`$_n$`)` lie in the set `in(p)`. If the answer is yes, we conclude that the **`proceed`** call holds the *same arguments* property.

To illustrate this analysis, consider the example **`around`** advice in Listing 5.1. As indicated by the comment at line 2, at the entry point, set of must reaching parameters is $\{$`(1,a),(2,b)`$\}$. The assign statement at line 5 generates the pair `(1,a2)`. Assuming that the **`proceed`** call at line 7 does not write fields of a, a2 and b, before line 9, the flow set remains.

```
1  void around$0(A a, B b)
2  { // { (1,a), (2,b) }
3     A a2, B b2;
4
5     a2 = a;
6     // { (1,a) (1,a2), (2,b) }
7     proceed(a2,b); // no write to a, a2 and b
8
9     if (cond)
10       b2 = b;
11       // { (1,a), (1,a2), (2,b), (2,b2) }
12    else
13       b2 = new B();
14       // { (1,a), (1,a2), (2,b) }
15    // { (1,a), (1,a2), (2,b) }
16    proceed(a,b2); // no write to a and b2
17
18    foo(a); // foo writes to a field of a
19    // { (2,b) }
20    proceed(a,b);
21 }
```

Listing 5.1 `UnchangedParamsAnalysis` example

Lines 9-16 illustrate a conditional and merge. On the true branch the pair `(2,b2)` is generated, but not on the false branch. After the merge, the intersection is computed as indicated in the comment on line 15, which does not include `(2,b2)`. Assuming that the **`proceed`** call at line 16 does not write fields of a, and b2, before line 18, the flow set remains.

Line 18 illustrates the use of side-effects to kill. If we assume that the call to `foo` writes to a field of `a`, then the side-effect analysis will indicate an interference with the pairs `(1,a)` and `(1,a2)` and these pairs are therefore killed. Since there is no loop, after line 20, the analysis reaches the fix point.

After that, we can test whether these **`proceed`** calls hold the *same arguments* property, and the results are below:

- The **`proceed`** at line 7 does have the same arguments (both `(1,a2)` and {(2,b) are in the set flowing in).

- The **`proceed`** at line 16 does not obey the same arguments property because `(2,b2)`

is not in the set flowing in.

- The **proceed** at Line 20 also does not obey the same arguments property because `(1,a)` is not the set flowing in.

UnchangedReturnAnalysis

In an **around** advice that returns a value, we must ensure that the value computed by a **proceed** actually reaches all return statements of the advice. `UnchangedReturnAnalysis` collects all values returned by an advice, and its result is used to check the *same return* condition. The specification of the data flow analysis is presented by the following six rules:

1. For each program point, we approximate a set of variables/expressions, represented by Soot's `Value`.
2. At a program point `p`, if `x` is in the set then this means that `x` either is returned by a return statement or has exactly the same value with a `Value` returned by a return statement. In particular, if the `x` is a reference type, then `x` is in the set at program point `p`, only if `x` refers to the same object as a `Value` returned by a return statement `ret`, and there has been no write to a field of the object between the `ret` and program point `p` or `ret` returns `x`.
3. It is a backward analysis.
4. The confluence operator is union:

$$out(n) = \bigcup_{s \in succ(n)} in(s)$$

   where: `out(n)` – means the set at the output of a node in the CFG, which is also the data set flowing into the node; `in(s)` – means the set at the input of a node in the CFG, which is also the data set flowing out of the node.

5. The data flow equation for a node `s` is

$$in(s) = out(s) \cap gen(s) - kill(s)$$

where the `out` and `in` have the same meaning with those in rule 4. The following are the `gen` and `kill` rules at a node:

- Gen due to return: a return statement in the form of `return op` generates `op`.
- Gen due to assign statement: if `lhs` of an assign statement in the form `lhs = rhs` is in the set, generate `rhs`.
- Kill due to assign statement: a statement of the form `lhs = rhs` kills `lhs`.
- Kill due to reference: for all statements of the form `a.f = b`, we kill `a`, and we generate a special value, `unknown`.
- Kill due to alias: if a value `v` is being killed, any value `v'` should be killed if `v` and `v'` point to the same location, *i.e.*, the `PointsToSet` of `v` and `v'` have a non-empty intersection.
- Kill due to side-effect: if the statement contains a method call, it may write to objects. Thus, for each value in the set, say `o`, we check (use Soot's side-effect analysis) to see if there in an interference with the method call and any field of `o`. If there is an interference, then we kill `o`, because the state of the object may have been changed, and it may no longer denote the same value being returned, and we generate a stub in the set to mark the value being returned is `unknown`.

6. Starting approximations are as follows:

- `in(end) =` {}.
- `in(other) =` {}.

The analysis is implemented under the backward data flow analysis framework in Soot. After the data-flow analysis reaches the fix point, we can query the data flow set to test the *same return value* property of **`proceed`** calls as follows: for an assign statement `p`

containing a **proceed** call — `x = proceed(...)`, we fetch out the data set flowing into `p`, *i.e.*,`out(p)`, and then check if `x` has the same value of all `Values` in the set `out(p)`. If the answer is yes, we conclude that the **proceed** call holds the *same return value* property.

```
1  int around$0()
2  {
3     int i;
4
5     x = proceed();
6     // { proceed() }
7     i = proceed();
8     // { i }
9     if (cond) {
10        // { i }
11        x = i;
12        // { x }
13        return x;
14    } else
15        // { i }
16        return i;
17 }
```

Listing 5.2 `UnchangedReturnAnalysis` example

To illustrate this analysis, consider the example **around** advice in Listing 5.2, and remember it is a backward analysis. The return statement at line 16 generates `i` to the set in the false branch, and the return at line 13 generates `x` to the set in the true branch. The assign statement at line 11 generates `i` and kills x. At the merge point at line 9, we union the sets from true and false branch, get {`i`}. At line 7, `i` is killed, but the expression `proceed()` is generated. After line 5, we reach the fix point.

Then, given the **proceed** at line 7, we can say it obeys the *same return value* property because `i` equals to `i`. However, given the **proceed** at 5, we would conclude that it violates the *same return value* property because `x` is not equal to `proceed()` (at least from the data flow analysis' view).

The example **around** advice in Listing 5.3 illustrates the analysis involving a reference type. Similarly, the return statement at line 12 generates `a` to the set in the false branch, and the return at line 9 generates `a` to the set in the true branch. However, the call to `foo` at line 7 changes `a`, so we kill `a` from the set and generate the stub `unknown`. At the merge point

```
A around$0()
{
   A a = proceed();
   // { unknown, a }
   if (cond) {
      // { unknown }
      foo(a); // foo writes to a field of a
      // { a }
      return a;
   } else
      // { a }
      return a;
}
```

Listing 5.3 `UnchangedReturnAnalysis` example demonstrating side-effect

at line 5, we union the sets from true and false branch, get {`unknown, a`}. After line 3, the analysis reaches the fix point. Then, given the **`proceed`** at line 3, we can conclude that it violates the *same return value* property because `unknown` does not equal `a`.

The above example also demonstrates the use of `unknown`, if we do not generate the stub when we kill `a` at line 7, at the merge point, we will get {`a`}, and then we will conclude that the **`proceed`** at line 3 obeys the same return value property, which is not the fact.

`ExceptionBeforeProceedAnalysis`

Although an unchecked exception thrown in an advice may terminate the execution of the advice, we only take checked exceptions into account in our analysis because every Java statement may throw unchecked exceptions, such as `OutofMemoryError`, which leads to too many possible (but unlikely) exceptions.

This analysis checks if there are uncaught checked exceptions thrown before a given **`proceed`** statement, and its result is used to check the *no abrupt exception* condition. Since we only care about checked exceptions, and since an **`around`** advice cannot throw an exception itself, it follows that any checked exception thrown in the body of the advice must be caught by an enclosing try-catch block. Thus, only statements in the same try-catch block with a **`proceed`** statement have the possibility to throw a checked exception that could cause the **`proceed`** statement to be bypassed. Therefore, in this analysis, we first try

to find if the **`proceed`** statement is contained inside a try-catch block, represented by `Trap` in Soot. If the answer is no, we can conclude that the *no abrupt exception* condition holds. If the answer is yes, we then conduct a data flow analysis to collect statements throwing checked exceptions. The analysis result is used to test if the *no abrupt exception* condition holds. The specification of the data flow analysis is presented by the following six rules:

1. For each program point, we approximate a set of statements, represented by Soot's `Stmt`.

2. At a program point `p`, if `x` is in the set then this means that the intersection set of the set of `Traps` surrounding `x` and the set of `Traps` surrounding the given **`proceed`** statement is not empty, and `x` throws checked exceptions.

3. It is a forward analysis.

4. The confluence operator is union:

$$in(n) = \bigcup_{s \in succ(n)} out(s)$$

   where: `out(s)` – means the set at the output of a node in the CFG, which is also the data set flowing out of the node; `in(n)` – means the set at the input of a node in the CFG, which is also the data set flowing into the node.

5. The data flow equation for a node `s` is

$$out(s) = in(s) \cap gen(s) - kill(s)$$

   where the `out` and `in` have the same meaning with those in rule 4. The following are the `gen` and `kill` rules at a node `s`:

   - Gen due to throw statement: if `s` is a direct throw statement and is surrounded by the same `Trap` and throws a checked exception, generate `s`.

- Gen due to method call: if `s` contains method invocation `m` and is surrounded by the same `Trap`, and `m` declares checked exceptions in its signature, generate `s`.
- Kill nothing.

6. Starting approximations are as follows:

- `in(end) = {}`.
- `in(other) = {}`.

The analysis is implemented under the forward data flow analysis framework in Soot. After the data-flow analysis reaches the fix point, we can query the data flow set to test the *no abrupt exception* property of the **`proceed`** calls as follows: for a statement `p` containing a **`proceed`** call, we fetch out the data set flowing into `p`, *i.e.*,`in(p)`, and then check if `in(p)` is empty. If the answer is yes, we conclude that the **`proceed`** call holds the *no abrupt exception* property.

5.2.2 Computation Impact

`ProceedAnalysis`

With the above three analyses, we implement the `ProceedAnalysis` to discover how proceed calls appear in control flow paths of an around advice.

First, we traverse all statements in the advice body looking for **`proceed`** calls. If the advice has void return type, which means the proceed call returns nothing, we apply `UnchangedParamsAnalysis` and the `ExceptionBeforeProceedAnalysis` on **`proceed`** call statements to check if the *same arguments* and *no abrupt exception* conditions are satisfied. If the advice has a return type, we also apply the `UnchangedReturn-Value` analysis to check if the *no abrupt exception* is satisfied. Then, we collect all exact-proceed calls into a set `exactProceedSet`, and all proceeds that are not exact-proceed into a set `proceedSet`. If neither set is empty, we run a data-flow analysis to check how these proceed calls are distributed on different control flow paths. The specification of the data flow analysis is presented by the following six rules:

1. For each program point, we approximate a set of flags. There are three different kinds of flags, represented by *NonePath* — which means there is a path having no proceed call at all, *NEPath* — which means there is a path having non-exact-proceed call but the path has no exact-proceed call, and *ExactPath* — which means there is a path having an exact-proceed call.

2. At a program point `p`, if flag `x` is in the set then this means that from start point to `p`, there exists a path described by `x`; if flag `y` is not in the set, this means that from start point to `p`, there does not exist a path described by `y`.

3. It is a forward analysis.

4. The confluence operator is union:

$$in(n) = \bigcup_{s \in succ(n)} out(s)$$

   where: `out(s)` – means the set at the output of a node in the CFG, which is also the data set flowing out of the node; `in(n)` – means the set at the input of a node in the CFG, which is also the data set flowing into the node.

5. The data flow equation for a node `s` is

$$out(s) = in(s) \cap gen(s) - kill(s)$$

   where the `out` and `in` have the same meaning with those in rule 4. The following are the `gen` and `kill` rules at a node `s`:

   - Gen *NonePath*: if `s` contains no proceed call and `in(s)` does not contain *NEPath* or *ExactPath*.
   - Gen *NEPath*: if `s` contains non-exact-proceed call and `in(s)` does not contain *ExactPath*.
   - Gen *ExactPath*: if `s` contains exact-proceed call.
   - Kill *NonePath*: if generating *NEPath* or *ExactPath*.

- Kill *NEPath*: if generating *ExactPath*.

6. Starting approximations are as follows:

   - `in(end) = {}`.
   - `in(other) = {}`.

This analysis is implemented under the forward data flow analysis framework in Soot. After the data-flow analysis reaches the fix point, we can query the data flow set to test how proceed calls are distributed on different control flow paths as follows: we fetch out the data set flowing into `end`, *i.e.*,`in(end)`, which has 6 possible values:

{*NEPath*}: which means for every path, there is at least one non-exact-proceed call but no exact-proceed call.

{*ExactPath*}: which means for every path, there is at least one exact-proceed call.

{*NonePath*, *NEPath*}: which means for every path, either it contains no proceed call, or it contains at least one non-exact-proceed call but no exact-proceed call.

{*NonePath*, *ExactPath*}: which means for every path, either it contains no proceed call, or it contains at least one exact-proceed call.

{*NEPath*, *ExactPath*}: which means for every path, either it contains at least one non-exact-proceed call but no exact-proceed call, or it contains at least one exact-proceed call.

{*NonePath*, *NEPath*, *ExactPath*}: which means some paths contain no proceed call; for some paths, they contains at least one non-exact-proceed call but no exact-proceed call; and for some paths, they contains at least one exact-proceed call.

Notice that {*NonePath*} is not possible since it means there is no proceed call at all on all paths, and the data flow analysis will not be called in this situation, but to be consistent, the analysis returns {*NonePath*} to represent that there is no proceed call.

Classify Computation Impact

Based on the results of the above analyses and the kind of the advice, we conclude computation impact using the following rules (also described in Algorithm 5.1:

Elimination Impact: If an **`around`** advice body is empty or only contains a return statement, it has elimination impact since the matched computation in the base program is totally removed.

Addition Impact: We categorize all non-empty **`before`** and **`after`** advice as causing addition impact. In addition, if an **`around`** advice has at least one exact-proceed on every path (`ProceedAnalysis` returns $\{$*ExactPath*$\}$) and has additional computation other than proceed calls, it has addition impact.

Definite-Substitution Impact: If an **`around`** advice does not have a proceed call on all paths(`ProceedAnalysis` returns $\{$*NonePath*$\}$) and also defines new computation in its body, the matched computation is definitely replaced by computation defined in advice, so the **`around`** advice has definite-substitution impact.

Conditional-Substitution Impact: If an **`around`** advice has at least one *exact-proceed* on one or more paths but not on all paths and has no **`proceed`** call at all on all other paths (`ProceedAnalysis` returns $\{$*NonePath*, *ExactPath*$\}$), the matched computation is conditionally replaced by computation defined in advice, so the **`around`** advice has conditional-substitution impact.

Mixed Impact: If an **`around`** advice does not have the above impact(`ProceedAnalysis` returns a set containing flag *NEPath*), it means on some paths, the matched computation may or may not be replaced by computation defined in advice at run-time, so the **`around`** advice has mixed impact.

Through our classification, programmers can get a general view of what an advice is going to do if applying it, or use our conclusion to verify if the advice is implemented as expected. For example, an advice is designed to totally replace the computation in the base program, but our analyses shows that the advice has conditional-substitution computation

impact. Noticing the difference between the intention and the analysis result, the programmer could become aware of the problem. Then, he/she just needs to focus on checking if there is an exact-proceed on some paths. Therefore, our analyses can also alleviate the difficulty of finding bugs in advice.

5.3 Experimental Results

5.3.1 Examples

First, we explore our analyses with the *bank* and *source code repository* examples discussed in Section 3.2.1.

Bank

A segment of our analyses report related to advice in the bank example is shown in Listing 4.1. For presentation purposes, we omit the part related to state impact. As expected, our analyses report that all three advice have addition computation impact. If we examine the `before` (Listing 3.4, line 6-11) and `after` (line 13-17) advice, they have addition computation impact because they are before/after advice and have non-empty bodies. If we examine the `around` advice (line 19-25), it has addition computation impact because it contains an exact-proceed (line 23) and extra computation.

Source Code Repository

Listing 5.5 shows the report of analyzing the source code repository example. As expected, our analyses classify the `boolean around` advice defined in SourceCodeRepositoryAspect.aj (Listing 3.13) at line 5-8 has elimination computation impact, and the `void around` advice at line 10-18 has conditional-substitution computation impact. If we examine the `boolean around` advice, it has elimination computation impact because it is an around advice having a return type other than void, and its body only contains a return statement. If we examine the `void around` advice, it has conditional-substitution computation impact because it has an exact-proceed (line 14) in one path, but no proceed at all on other paths.

```
AccountAspect.aj:6,1-11:2 Advice: before(bank.account.AbstractAccount
    account)
state impact:
...
addition computation impact

AccountAspect.aj:13,1-17:2 Advice: after(bank.account.AbstractAccount
    account)
state impact:
...
addition computation impact

AccountAspect.aj:19,1-25:2 Advice: void around()
no state impact
addition computation impact
```

Listing 5.4 Report of analyzing advice in the bank example

```
SourceCodeRepositoryAspect.aj:5,1-8:2 Advice: boolean around()
no state impact
elimination computation impact

SourceCodeRepositoryAspect.aj:10,1-18:2 Advice: void
    around(java.lang.String src)
no state impact
conditional-substitution computation impact
```

Listing 5.5 Report of analyzing the source code repository example

5.3.2 Benchmarks

Table 5.1 shows statistics of analyzing these benchmarks. The second column "C&A" shows the total number of classes and aspects. The third column shows the total number of advice declarations. The next five columns shows the number of advice declarations causing addition (**A**), elimination (**E**), definite-substitution (**D**), conditional-substitution (**C**) and mixed (**M**) computation impact respectively. The last column shows the number of classes and aspects checked when classifying these computation impacts manually. Similar to what we did for state impact, we started from the source code of all aspects and discovered all advice. Since all non-empty **before** and **after** advice are classified as having additional computation impact, we only need to check to see if their body is empty or not. For around

advice, we read its source code and determined its impact based on our classification. If a statement contains a method call, we had to discover the target methods and examined what those methods do by reading their source code. During this process, we recorded the number of aspects and classes visited.

benchmark	C&A	advice	A	E	D	C	M	C&A manually
aopbank	6	5	2	0	0	3	0	3
bean	3	2	2	0	0	0	0	2
DCM	34	8	7	0	0	0	0	4
exptree6	12	4	3	0	0	1	0	5
observer	8	1	1	0	0	0	0	2
ProdLine	20	15	13	0	1	0	1	11
Tetris	17	21	15	1	1	4	0	8
tracing	6	4	4	0	0	0	0	2

Table 5.1 Basic statistics about benchmarks

From Table 5.1, we can find that for most benchmarks, when classifying these computation impacts manually, nearly 50% of classes and aspects need to be checked. In addition, the source lines of code of advice declarations in these benchmarks are small, and the logic of these advice declarations are not complex. Therefore, we can image the difficulty of classifying and understanding computation impacts in a large-scale application. However, by using our tool, it only takes few minutes to get complete computation impact information.

We can find all categories of computation impact in these benchmarks. This gives us an intuition that our classification is reasonable. We also find 78% of advice causes addition computation impact. This reflects the reality that most of current AOP applications are in the first phase — "introducing exploration and enforcement aspects" [Duc06]. Aspects created in this phase usually do not modify the computation in the rest of the program.

In addition, for the tracing benchmark, we state that the aspects are pure observers when introducing it in Section 2.2. Thus, we would expect only addition impacts. As indicated in Table 5.1, our analysis gives the expected result.

Another interesting example is the around advice declaration causing definite-substitution computation impact in the ProdLine benchmark. In the base program, a `Graph` declares

an `addAnEdge(...)` method (in fact, this method is injected by an inter-type method declaration), which adds an edge to the graph but ignoring the weight information of the edge. This around advice crosscuts the invocation of the method and replace it with adding a weighted edge. Therefore, this advice is designed to replace old computations. As expected, our analysis reports definite-substitution computation impact.

Chapter 6
Shadowing Impact

In Section 3.5, we introduced the *shadowing impact* and defined it as the change of reference of fields in the *base program*, which refers to the whole program except the inter-type declaration being considered, after applying inter-type declarations. Although there are various kinds of inter-type declarations, such as inter-type field, method, constructor, parents, warning, error, soft, precedence declaration, in AspectJ, it is obvious that only two of them can cause a shadowing impact, which are *inter-type field declaration*s and *inter-type parents declaration*s, and two different analyses are implemented to deal with them respectively. Our analyses report whether there is shadowing impact on `[Class].field`[1] if there is such a field reference somewhere in the application. Extending this to identify individual reference sites is very straightforward, and the basic idea is simple: for each field reference in the application, check if the receiver's type and the name of field match to one in our shadowing impact set. In this chapter, we discuss *shadowing impact* and the two analyses in detail.

[1] In this thesis, we denote the access of field `f` of a variable of type `Type` as `[Type].f`.

6.1 Inter-type Field Declaration

6.1.1 Definition

An inter-type field declaration has the form of "`Type TargetClass.foo`", and it injects a new field named "`foo`" of type `Type` into the class `TargetClass`.

It may cause a shadowing impact on `TargetClass` if there is a field also named "`foo`" in a super class of `TargetClass` (*e.g.*,`Super.foo`), and that field is inheritable by `TargetClass`. The newly declared field `TargetClass.foo` will shadow `Super.foo`, which means a reference `[TargetClass].foo` refers to different field after applying `Type TargetClass.foo`, *i.e.*, it refers to `Super.foo` before applying the inter-type declaration and to `TargetClass.foo` after applying the inter-type declaration.

In addition, if `Type TargetClass.foo` causes a shadowing impact on `TargetClass`, it also causes a shadowing impact on subclasses of `TargetClass` except for the set `s` of subclasses that declare their own "`foo`" field and all subclasses of classes in `s`. In the same way, "`[subclasses].foo`" matches to `Super.foo` and `TargetClass.foo`, respectively, before and after applying `Type TargetClass.foo`.

Therefore, for each field "`foo`" that is involved in a shadowing impact, three kinds of information matters:

affected types: are all classes affected by the shadowing impact, *i.e.*, `TargetClass` itself and all its subclasses except for the set `s` of subclasses that declare their own "`foo`" field and all subclasses of classes in `s`.

original type: is the type that declares the field to which `[affected types].foo` refers before applying the inter-type declaration, *i.e.*,`Super`.

current type is the type that declares the field to which `[affected types].foo` refers after applying the inter-type declaration, *i.e.*,TargetClass.

6.1.2 Analysis

To discover if there is shadowing impact, our analysis checks if a new declared field shadows a field which is inherited by `Class` from its super classes and has the same name. At the conceptual level, our algorithm is presented in Algorithm 6.1.

Algorithm 6.1: Shadowing impact analysis on inter-type field declaration

```
1  foreach inter-type field declaration target.id in the application do
       /* check whether shadowing impact is caused                    */
2      put parent of target into worklist1;
3      while worklist1 is not empty and not hasimpact do
4          get first element c from worklist1;
5          foreach field f declared in c do
6              if f.name == "id" then
7                  hasimpact;
8                  record c as originalType;
9                  put target into affectedTypes;
10                 record target as currentType;
11             end
12         end
13         if c != java.lang.Object then add parent of c into worklist1;
14     end
       /* discover all affected types                                 */
15     if hasimpact then
16         put children of target into worklist2;
17         while worklist2 is not empty do
18             get first element c from worklist2;
19             foreach f declared in c do
20                 if f.name == "id" then redefined;
21             end
22             if !redefined then
23                 put c into affectedTypes;
24                 add children of c into worklist2;
25             end
26         end
27     end
28 end
```

First, we fetch all inter-type field declarations from abc. Then, for each inter-type field declaration having the form `target.id`, we check if it causes a shadowing impact. Basically, we first traverse the class hierarchy up starting from `target`(exclusive) to find if "`id`" is declared there. If yes, we say that `target.id` causes shadowing impact, and record `c` as original type, `target` as current type, and also consider `target` as an affected type. Then, we traverse the class hierarchy down starting from `target`(exclusive) to find and record subclasses affected by this impact, which are all subclasses except those subclasses who declare "`id`" and their subclasses. Therefore, during the analysis process, *original type*, *current type* and *affected types* are analyzed and recorded. As a result, in our report, we not only report whether there is shadowing impact, but also what the change is.

To illustrate our algorithm, consider the example hierarchy in Figure 6.1. `B.id` is an inter-type field declaration. Traversing up in the hierarchy, we found `A` declares field "`id`", thus `B.id` causes shadowing impact, and we record `A` as the original type and `B` as the current type, and also put `B` into the affected types set. Then, we traverse down in the hierarchy, and we reach `C` and `E`, children of `B`. `C` does not redefine "`id`", so we add it into affected types set and add `D`, the child of `C`, into the `worklist2`. `E` redefines "`id`", so we stop traversing this path by not expanding the `worklist2`. After that, we get `D` from `worklist2` and add `D` into the affected types set similarly. Once the worklist is empty, the analysis ends. As a result, we conclude that there is shadowing impact regarding field "`id`", and record:

affected types: `B`, `C`, and `D`;

original type: `A`;

current type: `B`.

6.2 Inter-type Parents Declaration

There are two different kinds of inter-type parents declarations: *inter-type-extends-declaration* in the form of "`TypePattern` **`extends`** `Class`" and *inter-type-implements-*

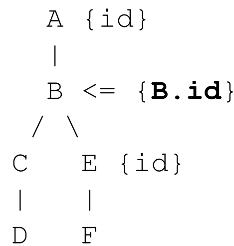

Figure 6.1 Shadowing impact analysis on inter-type field declaration example

declaration in the form of "`TypePattern` **`implements`** `Interface`". The inter-type-extends-declaration can cause shadowing impacts; however, the inter-type-implements-declaration would not cause any shadowing impact at all. In the following sections, we will discuss them and the corresponding analysis in detail.

6.2.1 Inter-type-extends-declaration

Definition

The inter-type-extends-declaration only allows a class `c` to extend its siblings or its siblings' sub-classes. This restriction means a class can only be moved down in the hierarchy through its siblings path, thus only new fields can be inherited by the class, and it can not lose any fields. For example, in the class hierarchy shown in Figure 6.2, `C extends E`, `C extends F`, and `E extends C` are allowed, but `C extends A` and `F extends C` are illegal.

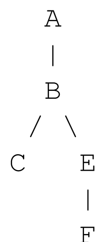

Figure 6.2 An example class hierarchy

An inter-type-extends-declaration having the form of "`TargetClass extends NewParent`" can cause shadowing impact on a field named "`foo`" if both of the following two conditions are satisfied:

1. Some old super classes of TargetClass (*e.g.*, OldSuper) — which are super classes of TargetClass before applying the inter-type-extends-declaration, for example classes A and B if we declare C extends F — declare a field named "foo", and the field is inheritable by TargetClass.

2. Some new super classes of TargetClass (*e.g.*, NewSuper) — which are super classes newly extended after applying the inter-type extends declaration, for example classes E and F if we declare C extends F — declare a field named "foo", and the field is inherited by TargetClass.

Thus, the new inherited "foo" shadows the original inherited "foo. Here, OldSuper is the *original type*, NewSuper is the *current type*, and TargetClass is an *affected type*.

Similar to the inter-type field declaration, the inter-type-extends-declaration also causes shadowing impact on subclasses of TargetClass except for the set s of subclasses that declare their own "foo" field and all subclasses of classes in s. These subclasses are *affected type*s too.

All information regarding *current type*, *original type* and *affected types* are also recorded and reported in a similar way as for inter-type field declarations.

Analysis

From the above definition, we can see that there are some commonalities between inter-type field and extends declarations, and the analysis shares those commonalities. In our analysis, we first need to know what fields are newly inherited by the class, and then treat each of them like a new declared field similar to the field declared by inter-type field declaration. Those fields which are newly inherited actually come from fields that are declared in classes along the path between the old parent(exclusive) and the new parent(inclusive), so we need the old parent information. Therefore, our analysis is divided into two parts: the first part records the old parent before abc processes inter-type declarations, and the second part analyzes the shadowing impact after inter-type declarations are woven.

Part 1: record old parent

This part is shown in Figure 6.3 (which is the same as Figure 2.2), the box labeled "Hierarchy storing". For each inter-type-extends-declaration, we simply record its parent at that time. However, our *base program* refers to the whole program except the inter-type declaration being considered, so it means when we talk about the old parent of an inter-typed class, we should take the effect of other inter-type parents declarations into account. However, would other inter-type parents declarations interfere with the one being considered? The answer is no. If more than one inter-type-extends-declarations are defined in an application, either a) all new parents lie in the same path in the hierarchy tree; or b) new parents spread into different paths in the hierarchy tree. In situation a, only the one that declares the new parent deepest down into the hierarchy is applied, thus no interference so caused; in situation b, the AspectJ compiler will complain of a compile-error. Situation a is shown in Figure 6.4, both "`C extends E`" and "`C extends F`" are defined, but since `F` is deeper than `E` in the hierarchy, only "`C extends F`" is applied. In fact, abc gives a warning regarding such a situation. Therefore, we can simply record the parent before abc processes any inter-type declarations (*e.g.*,`B`) as the old parent.

Part 2: analysis after weaving

In an inter-type-extends-declaration, the target could be a type pattern, which may match to more than one class. Thus, we first fetch all target classes from abc, and for each target class, perform the following two steps.

The first step is to find all fields newly inherited by the target class, *i.e.*, the class being declared parents, *e.g.*, class `C` in the example in Figure 6.4. We first fetch out the old parent stored before weaving. Then, we traverse the hierarchy up starting from the new parent till the old parent to find the path connecting the old and new parent. In Figure 6.4, the path is `F–E`. After that, we traverse the hierarchy down through the path, `E–F`, and we record all fields declared in classes along the path and inheritable by the target class. In case that a class down in the path declares a field having the same name with a field declared in a class above it, and both fields are inheritable by the target class, only the field declared in

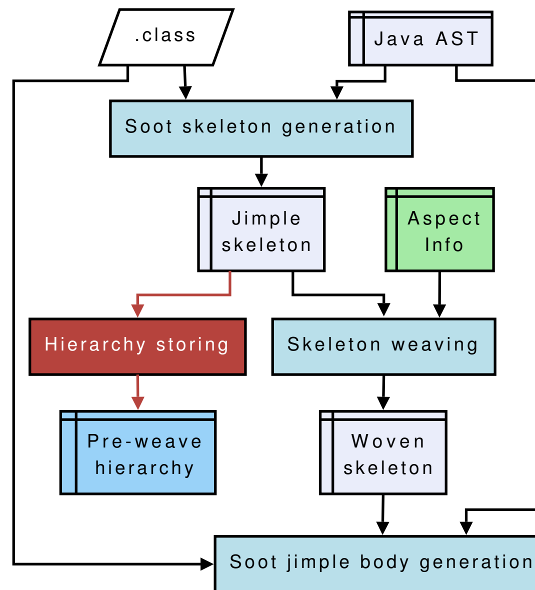

Figure 6.3 Code generation and static weaving of abc, and the pre-weave hierarchy recording phase of **AIA** in static weaving, extended from [ACH+05]

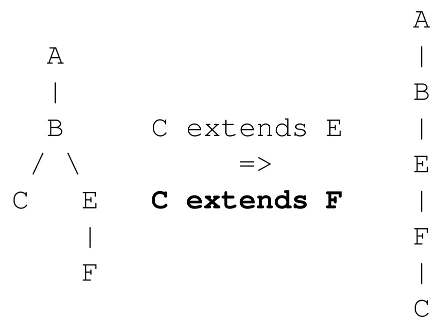

Figure 6.4 Two inter-type-extends-declarations on the same type example

the class further down in the path is recorded. For example, if both `E` and `F` declare a field "`sn`", only the "`sn`" in `F` is recorded.

In the second step, for each field recorded in the first step, we treat it as if it is a newly declared field by inter-type field declaration and following the similar algorithm as Algorithm 6.1, except that when traversing the hierarchy up, we skip classes between the old parent and the new parent (*e.g.*, `E` and `F`) and start from the old super class (*e.g.*, `B`).

6.2.2 Inter-type-implements-declaration

In fact, inter-type-implements-declaration does not cause any shadowing impact at all. If the new super interface declaring a field having the same name with a field inherited by or declared in the target class, *i.e.*, the class being declared new super interface, there are only two possible outcomes: a) compile error, the compiler complains "ambiguous field"; or b) there is no compile-time error, if the field is not reference anywhere in the application, but since it is not referenced at all, there will never be shadowing impact. As illustrated in Figure 6.5, if there is `[E].f` somewhere in the application, compiler will complain about an ambiguous field on `[E].f`. Therefore, there is no possibility for an inter-type-implements-declaration causing shadowing impact.

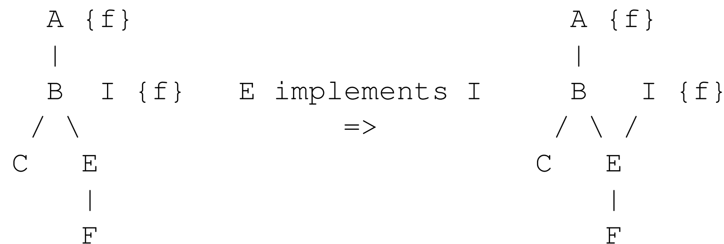

Figure 6.5 Inter-type implements declarations example

6.3 Experimental Results

6.3.1 Examples

First, we explore our analyses with the *bank* example discussed in Section 3.2.1.

Bank

A segment of our analyses report related to inter-type declarations in the bank example is shown in Listing 6.1. For presentation purposes, we omit the part related to lookup impact. As expected and discussed in Section 3.5, our analyses report that both the inter-type field declaration **`GoldCard.bonus`** (Listing 3.9, line 5) and the inter-type parents declaration `ValueCard extends RewardCard` (line 7) cause shadowing impacts.

```
CreditCardAspect.aj:5,8-29 ITD: GoldCard.bonus
shadowing impact:
   [bank.creditcard.GoldCard].bonus
      originally matched to bank.creditcard.RewardCard, currently
         matches to bank.creditcard.GoldCard

CreditCardAspect.aj:9,8-61 ITD:
   GoldCard.payment(bank.account.StudentCheckingAccount)
lookup impact:
...

CreditCardAspect.aj:7,1-47 ITD: ValueCard extends RewardCard
shadowing impact:
   [bank.creditcard.ValueCard].bonus
      originally matched to bank.creditcard.AbstractCreditCard,
         currently matches to bank.creditcard.RewardCard
lookup impact:
...
```

Listing 6.1 Report of analyzing inter-type declarations in the bank example

The report shows the inter-type field declaration **GoldCard.bonus** causes a change of binding for the field reference [GoldCard].bonus. In addition, the analyses report that [GoldCard].card refers to `RewardCard.bonus` before applying the inter-type field declaration and refers to `GoldCard.bonus` instead after applying the inter-type field declaration.

The report also shows that the inter-type parents declaration **ValueCard extends RewardCard** causes a binding change for the field reference [ValueCard].bonus. Similarly, the analyses report that [ValueCard].bonus refers to `AbstractCreditCard.bonus` before applying the inter-type parents declaration and refers to `RewardCard.bonus` instead after applying inter-type parents declaration.

6.3.2 Benchmarks

Table 6.1 shows statistics of analyzing these benchmarks. The second column “C&A” shows the total number of classes and aspects. The third column shows the total number of inter-type declarations. The next column shows the number of shadowing impacts. The last column shows the number of classes and aspects checked when discovering these

shadowing impact manually. Basically, we started from the source code of all aspects and discovered all inter-type declarations. For each inter-type declaration, we tried to apply it to the base program manually by drawing the partial UML graph[2], and then examined the hierarchy of the UML graph to determine if there is shadowing impact caused. During this process, we recorded the number of aspects and classes visited, including the classes that have to be visited to determine the hierarchy.

benchmark	C&A	inter-type declarations	shadowing	C&A manually
aopbank	6	1	0	3
bean	3	7	0	2
DCM	34	2	0	4
exptree6	12	27	0	5
observer	8	11	0	2
ProdLine	20	79	0	20
Tetris	17	0	0	8
tracing	6	0	0	2

Table 6.1 Basic statistics about benchmarks

In fact, we can't find any shadowing impact in these benchmarks. We think the reason is that shadowing impact usually involves non-private fields, and as the encapsulation discourages non-private fields, thus the chance of shadowing impact happening is lower than other kinds of impact. Moreover, shadowing impact usually leads to bugs, and these benchmarks are well tested, and this reduces the chance of finding shadowing impact further.

Nevertheless, to determine manually that there is no shadowing impact, we would still need to check nearly 50% percent of classes and aspects. For ProdLine, we have to check all classes and aspects since inter-type declarations are heavily used in it. Thus, we can image the difficulty of discovering and understanding shadowing impacts in a large-scale application. However, by using our tool, it only takes few minutes to get complete shadowing impact information.

In addition, for the tracing benchmark, we state that the aspects are pure observers when introducing it in Section 2.2. From the above table, we can see that as expected there is no

[2]Since only fields are concerned, we did not draw methods, so we call it partial UML graph.

shadowing impact caused.

Chapter 7
Lookup Impact

In Section 3.6, we introduced the concept of *lookup impact* and defined it as an impact caused by changing the lookup of a method invocation in the *base program* (which refers to the whole program except the inter-type declaration being considered) after applying an inter-type declaration. Although there are various kinds of inter-type declarations, such as inter-type field, method, constructor, parents, warning, error, soft, precedence declaration, in AspectJ, it is obvious that only three of them can cause a lookup impact, which are *inter-type method declaration*, *inter-type constructor declaration* and *inter-type parents declaration*; three different analyses are implemented to deal with them respectively. Similar to shadowing impact analyses, we report whether there is a lookup impact on `[Class].method(...)` [1] if there is such a method invocation somewhere in the application. Extending this to identify individual call-site is not difficult and has already been implemented by one of our graduate students taking "Optimizing Compilers (Winter 2008)" course as a course project. In this chapter, we discuss *lookup impact* and the three analyses in detail.

7.1 Finding the Matched Method

Since we define lookup impact as causing a change of lookup of a method invocation, the lookup impact analysis involves finding the most-specific method for a method invocation.

[1] In this thesis, we denote a method invocation `foo()` on a receiver of type `Type` as `[Type].foo()`.

In this section we briefly discuss the most-specific matching strategy. In addition, the matched method of a method invocation of the same method on receivers of the same type may be different depending on where the invocation appears, thus we also introduce the *invocation place* concept in this section.

7.1.1 Accessible Methods and Invocation Place

Whether a method declaration is accessible at a method invocation depends on the access modifier (public, none, protected, or private) in the method declaration and on where the method invocation appears. Therefore, the place of invocation should be considered in our analysis. Guided by the Java Language Specification [GJSB05] (**JLS**) 6.6 — Access Control, we define four different kinds of *invocation places*:

class: means the invocation appears within the class of the receiver, .

package: means the invocation appears within the package of the receiver's class but not within the receiver's class.

protected: means the invocation appears within a class that is a super class of the receiver's class that either declares or inherits the invoked protected method, but not within the receiver's package.

other: means the invocation appears somewhere other than the above three places.

Consider the class hierarchy in Figure 7.1, for the invocation `[D].foo(LinkedHash-Set)`:

Within class means within `p3.D`.

Within package means within `p3.E`.

Within protected means within `p4.C` or `p2.B`.

Within other means within `p1.A` or `p4.F`.

```
p1.A {public foo(Object)}    p3.E   p4.F
     {public foo(int)}
 |
p2.B {protected foo(Set)}
 |
p4.C
 |
p3.D {private foo(LinkedHashSet)
      foo(HashSet)}
```

Figure 7.1 Invocation place example

For a method invocation `[Type].foo(...)` appearing in an invocation place, we identify *accessible methods* of the invocation as all methods that are declared or inherited by `Type` and are permitted to access in the invocation place according to **JLS** 6.6 — Access Control. Basically, they are visible methods based on the invocation place among all methods that are declared or inherited by `Type`.

For the invocation `[D].foo(LinkedHashet)` on the class hierarchy in Figure 7.1, accessible methods are described as follows:

Within class, accessible methods are `p1.A.foo(Object)`, `p1.A.foo(int)`, `p2.B.foo(Set)`, `p3.D.foo(HashSet)` and `p3.D.foo(LinkedHashSet)`.

Within package, accessible methods are `p1.A.foo(Object)`, `p1.A.foo(int)` and `p3.D.foo(HashSet)`.

Within protected, accessible methods are `p1.A.foo(Object)`, `p1.A.foo(int)` and `p2.B.foo(Set)`.

Within other, accessible methods are `p1.A.foo(Object)` and `p1.A.foo(int)`.

7.1.2 Applicable Methods

Amongst all accessible methods, only those that satisfy the conditions as described in **JLS** 15.12.2.1 — Identify Potentially Applicable Methods — are *applicable methods*. Basically, they are methods that have the identical name and method invocation convertible formals

with the invoked method. For the above example, `p1.A.foo(int)` is not applicable, since `int` and `LinkedHashSet` is not method invocation convertible, thus:

Within class, applicable methods are `p1.A.foo(Object)`,`p2.B.foo(Set)`,`p3.D.foo (HashSet)` and `p3.D.foo(LinkedHashSet)`.

Within package, applicable methods are `p1.A.foo(Object)` and `p3.D.foo(HashSet)`.

Within protected, applicable methods are `p1.A.foo(Object)` and `p2.B.foo(Set)`.

Within other, applicable methods are `p1.A.foo(Object)`.

7.1.3 Most Specific Method

If more than one method are both accessible and applicable to a method invocation, Java uses the most specific matching strategy to choose the matched method, and this is discussed in detail in **JLS** 15.12.2.5 — Choosing the Most Specific Method. "The informal intuition is that one method is more specific than another if any invocation handled by the first method could be passed on to the other one without a compile-time type error." However, it is possible that no method is the most specific. For example, for an invocation `[Type].foo(HashSet, HashSet)`, assume two methods are applicable: `foo(Set, HashSet)` and `foo(HashSet, Set)`; however, neither one is more specific than the other, so there is no most specific method.

For the invocation `[D].foo(LinkedHashet)` on the class hierarchy in Figure 7.1, the most specific method is described as follows:

Within class, the most specific methods is `p3.D.foo(LinkedHashSet)`.

Within package, the most specific methods is `p3.D.foo(HashSet)`.

Within protected, the most specific methods is `p2.B.foo(Set)`.

Within other, the most specific methods is `p1.A.foo(Object)`.

7.2 Inter-type Method Declaration

7.2.1 Definition

An inter-type method declaration has the form of "RetType TargetClass.foo(formals){...}", and it injects a new method "RetType foo(formals)" into the class TargetClass.

It may cause a lookup impact on [TargetClass].foo(formals) if this invocation matches to a method C.foo(...) before applying the inter-type method declaration since [TargetClass].foo(formals) will match to the newly declared method TargetClass.foo(...). However, the method originally matched may be different for different invocation places, thus the analysis should consider each invocation place respectively.

In addition, no matter if RetType TargetClass.foo(formals){...} causes lookup impact on TargetClass, it may cause lookup impact on subclasses of TargetClass except for the set s of subclasses that declare their own foo(formals) and all subclasses of classes in s. However, due to the most specific match, each subclass should be considered individually to determine the method that [subclass].foo(formals) originally matches to. Similarly, each invocation place should be considered respectively.

Therefore, to report a lookup impact, we should report all affected types, which are defined as follows:

affected types: are all classes affected by a lookup impact, *i.e.*,, TargetClass itself and all its subclasses except for the set s of subclasses that declare their own foo(formals) and all subclasses of classes in s.

For each affected type, for each possible invocation place, we should report the original method and current method, which are defined as follows:

original method: is the most-specific method that [affected type].foo(formals) originally matches to before applying the inter-type declaration.

current method: is the most-specific method that `[affected type].foo(formals)` currently matches to after applying the inter-type declaration.

7.2.2 Analysis

Guided by Java Language Specification (**`JLS`**), our analysis is presented in Algorithm 7.1.

An inter-type method declaration can declare that interfaces have methods with bodies, but in abc (also in ajc), these methods are actually woven into the interface's direct implementors; thus, this kind of declaration is equivalent to a number of inter-type method declarations declared on each of these implementors. Therefore, at line 3 in our algorithm (Algorithm 7.1), we put all direct implementors into the `worklist` and then analyze them as if they are inter-type method declaration on classes.

For each class in the worklist, for each kind of invocation place, we first try to find the most specific method to determine if lookup impact is caused at line 10-19. If the most specific method exists, we can conclude that lookup impact is caused and record the most-specific method as the original method, and obviously the newly injected method is the current method, and we also record the class being analyzed `c` as an affected type for this kind of invocation place.

No matter if a lookup impact is caused on class `c` or not, the inter-type method declaration may cause lookup impact on its subclasses due to the most specific matching strategy unless the subclass overrides the newly declared method. Therefore, we add all `c`'s direct subclasses that do not override the newly declared method into the worklist so that they can be analyzed. By using the worklist design pattern, all subclasses of `c` will be analyzed in the following loops.

To illustrate our algorithm, consider the example hierarchy in Figure 7.2. `public B.foo(HashSet) {...}` is an inter-type method declaration. First, we analyze `B`.

```
    A {public foo(Object)}
    |
    B {private foo(Set)} <= {public B.foo(HashSet){...}}
   / \
  C   E {public foo(HashSet)}
```

Figure 7.2 Lookup impact analysis on inter-type method declaration example

Algorithm 7.1: Lookup impact analysis on inter-type method declaration

```
1  foreach inter-type method declaration class.id(formals) in the application do
2      if class is interface then
3          put direct implementors of class into worklist;
4      else
5          put class into worklist;
6      end
7      while worklist is not empty do
8          get first class c from worklist;
9          foreach invocation place p do
10             hasLookup = false;
11             put all methods inherited by c into mset;
12             put member methods of c into mset except id.(formals);
13             remove not accessible methods based on p from mset;
14             remove not applicable methods from mset;
15             if size of mset == 1 then hasLookup = true;
16             else
17                 sort mset ordered by more specific property;
18                 if mset.get(1) is more specific than mset.get(2) then
19                     hasLookup = true;
20                 end
21             end
22             if hasLookup then
23                 record c as affectedType;
24                 record the first method in mset as originalMethod;
25                 record class.id(formals) as currentMethod;
26             end
27         end
28         foreach direct subclass child of c do
29             if child do not declare id.(formals) then
30                 add child into worklist;
31             end
32         end
33     end
34 end
```

For the *class* invocation place, the accessible methods are `A.foo(Object)` and `B.foo(Set)`, and the applicable methods are the same. After sorting by the more specific property, they are ordered as `B.foo(Set)` and `A.foo(Object)`. The first element `B.foo(Set)` is more specific than the second element `A.foo(Object)`; thus, `B.foo(Set)` is the most specific method, and a lookup impact is caused. Therefore, for invocation place class, `B` is affected; `B.foo(Set)` and `B.foo(HashSet)` are the original method and current method respectively.

For the *package* invocation place, the accessible methods are `A.foo(Object)`, and the applicable methods are the same. Since there is only one applicable method, it is the most specific method, and a lookup impact is caused. Therefore, for invocation place package, `B` is affected; `A.foo(Object)` and `B.foo(HashSet)` are the original method and current method respectively.

For the *protected* and *other* invocation place, if we follow the algorithm, we get the same result as for package invocation place.

Then, we analyze class `C`. For all kinds of invocation place, we get `C` is affected, and the original method and current method are `A.foo(Object)` and `B.foo(HashSet)` respectively.

The algorithm will skip `E` since it declares its own `foo(HashSet).`

Table 7.1 summarizes the information recorded.

affected type	invocation place	original method	current method
`B`	*class*	`B.foo(Set)`	`B.foo(HashSet)`
`B`	*package*	`A.foo(Set)`	`B.foo(HashSet)`
`B`	*protected*	`A.foo(Set)`	`B.foo(HashSet)`
`B`	*other*	`A.foo(Set)`	`B.foo(HashSet)`
`C`	*class*	`A.foo(Set)`	`B.foo(HashSet)`
`C`	*package*	`A.foo(Set)`	`B.foo(HashSet)`
`C`	*protected*	`A.foo(Set)`	`B.foo(HashSet)`
`C`	*other*	`A.foo(Set)`	`B.foo(HashSet)`

Table 7.1 Analysis result of inter-type method declaration example

7.3 Inter-type Constructor Declaration

7.3.1 Definition

An inter-type constructor declaration has the form of "`TargetClass.new(formals) {...}`", and it injects a new constructor "`TargetClass(formals)`" into the class `TargetClass`.

It may cause a lookup impact on `new TargetClass(formals)` if this instantiation matches to a constructor `TargetClass(...)` before applying the inter-type constructor declaration since `new TargetClass(formals)` will match to the newly declared constructor `TargetClass(formals)` after applying the inter-type constructor declaration. However, the constructor originally matched may be different for different invocation places, thus the analysis should consider each invocation place respectively.

Unlike methods, constructors are not inheritable; thus, inter-type constructor declaration can't cause lookup impact on subclasses of `TargetClass`.

Similar to inter-type method declaration, to report the lookup impact, we should report all *affected types*. Similarly, for each *affected type*, for each possible *invocation place*, we should report the *original constructor* and *current constructor*.

7.3.2 Analysis

Without the issue of inheritance, the algorithm is simpler than the one for inter-type method declaration and is described in Algorithm 7.2.

We neither consider constructors of `c`'s super classes when gathering accessible constructors nor analyze subclasses of `c`.

To illustrate our algorithm, consider the example hierarchy in Figure 7.3. `public A.new(HashSet) {...}` is an inter-type constructor declaration.

```
A {public A(Object)} <= {public A.new(HashSet){...}}
  {private A(Set)}
```

Figure 7.3 Lookup impact analysis on inter-type constructor declaration example

For *class* invocation place, the accessible constructors are `A(Object)` and `A(Set)`,

Algorithm 7.2: Lookup impact analysis on inter-type constructor declaration

```
1  foreach inter-type constructor declaration c.new(formals) in the application
   do
2      foreach invocation place p do
3          hasLookup = false;
4          put constructors of c into mset except c(formals);
5          remove not accessible constructors based on p from mset;
6          remove not applicable constructors from mset;
7          if size of mset == 1 then hasLookup = true;
8          else
9              sort mset ordered by more specific property;
10             if mset.get(1) is more specific than mset.get(2) then
11                 hasLookup = true;
12             end
13         end
14         if hasLookup then
15             record c as affectedType;
16             record the first constructor in mset as originalCon;
17             record c(formals) as currentCon;
18         end
19     end
20 end
```

and the applicable constructors are the same. After sorting by the more specific property, they are ordered as `A(Set)` and `A(Object)`. The first element `A(Set)` is more specific than the second element `A(Object)`; thus, `A(Set)` is the most specific constructor, and lookup impact is caused. Therefore, for invocation place class, `A` is affected; `A(Set)` and `A(HashSet)` are original constructor and current constructor respectively.

For *package* invocation place, the accessible constructors are `A(Object)`, and the applicable constructors are the same. Since there is only one applicable constructor, it is the most specific constructor, and lookup impact is caused. Therefore, for invocation place package, `A` is affected; `A(Object)` and `A(HashSet)` are original constructor and current constructor respectively.

For *all* invocation place, if we follow the algorithm, we get the same result as for *pack-*

age invocation place.

7.4 Inter-type Parents Declaration

There are two different kinds of inter-type parents declarations: *inter-type-extends-declaration* in the form of "`TypePattern` **`extends`** `Class`" and *inter-type-implements-declaration* in the form of "`TypePattern` **`implements`** `Interface`". The inter-type-extends-declaration can cause lookup impacts; however, the inter-type-implements-declaration would not cause any lookup impact at all. In the following sections, we discuss them and the corresponding analysis in detail.

7.4.1 Inter-type-extends-declaration

Definition

As discussed in Section 6.2.1, the inter-type-extends-declaration only allows a class `c` to extend its siblings or its siblings' sub-classes. This restriction means a class can only be moved down in the hierarchy through its siblings path, thus only new methods can be inherited by the class, and it can not lose any methods.

An inter-type-extends-declaration having the form of "`TargetClass extends NewParent`" can cause a lookup impact on a method call `[Type].foo(formals)` in an invocation place if both of the following two conditions are satisfied:

1. The method invocation matches to a method `foo(...)` declared by a old super class of `TargetClass`(*e.g.*, `OldSuper`).

2. The method invocation matches to a method `foo(...)` declared by a new super class of `TargetClass`(*e.g.*, `NewSuper`).

Thus, `Type` is an affected type in this invocation place. The newly inherited method `NewSuper.foo(...)` is the *current method*, and the method `OldSuper.foo(...)` is the *original method*.

Similar to the inter-type method declaration, the inter-type-extends-declaration also causes a lookup impact on subclasses of `TargetClass` except for the set `s` of subclasses that declare their own "`foo(formals)`" and all subclasses of classes in `s`. These subclasses are *affected types* too. Due to the most specific strategy, similarly, all subclasses should be analyzed individually.

For each affected type, for each kind of invocation place, the information regarding original method and current method are also recorded and reported in a similar way as for inter-type method declarations.

Analysis

From the above definition, we can see that there are some commonalities between inter-type method and extends declarations and also between lookup and shadowing impact caused by inter-type-extends-declaration, and the analysis shares these commonalities. Similar to shadowing impact analysis, in this analysis, we first need to know what methods are newly inherited by the class, and treat each of them like a new declared method similar to the method declared by inter-type method declaration. Since constructors are not inherited, we can ignore them when dealing with inter-type-extends-declaration. Similar to shadowing impact analysis, those methods which are newly inherited actually come from methods that are declared in classes along the path between the old parent(exclusive) and the new parent(inclusive), so we need the old parent information. Therefore, this analysis is also divided into two parts: the first part records the old parent before abc processes inter-type declarations, and the second part analyzes the shadowing impact after inter-type declarations are woven.

Part 1: record old parent

This part is shown in Figure 6.3, the box labeled "Hierarchy storing". As described in Section 6.2.1, for each inter-type-extends-declaration, we simply record its parent at that time.

Part 2: analysis after weaving

In an inter-type-extends-declaration, the target could be a type pattern, which may match to more than one class. Thus, we first fetch all target classes from abc, and for each target class, perform the following two steps.

The first step is to find all methods newly inherited by the target class, *i.e.*, the class being declared parents, *e.g.*, class `C` in the example in Figure 7.4. We first fetch out the old parent stored before weaving. Then, we traverse the hierarchy up starting from the new parent till the old parent to find the path connecting the old and new parent. In Figure 7.4, the path is `F`–`E`. After that, we traverse the hierarchy down through the path, `E`–`F`, and we record all methods declared in classes along the path and inheritable by the target class. In case that a class down in the path overrides a method declared in a class above it, and both methods are inheritable by the target class, only the overriding method declared in the class down in the path is recorded. For example, if both `E` and `F` declare a method `sn()`, only the `sn()` in `F` is recorded.

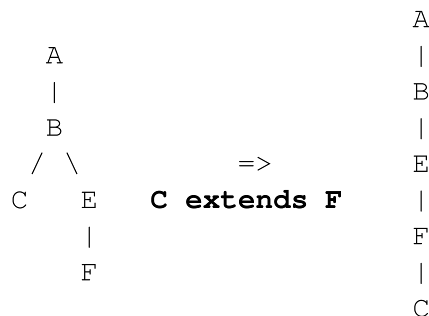

Figure 7.4 Inter-type-extends-declarations example

In the second step, for each method recorded in the first step, we treat it as if it is a newly declared method by inter-type method declaration and following the similar algorithm as Algorithm 7.1, except that when traversing the hierarchy up, we skip classes between the old parent and the new parent (*e.g.*, `E` and `F`) and start from the old super class (*e.g.*, `B`).

7.4.2 Inter-type-implements-declaration

In fact, the inter-type-implements-declaration does not cause any lookup impact at all. Since an interface can't declare a method with a body, for an inter-type-implements-declaration `TargetClass implements I`, methods declared in `I` either a) are already defined by `TargetClass`, or b) are injected by inter-type method declaration to `TargetClass` or `I`. For situation a), obviously, the inter-type-implements-declaration will not cause any lookup impact. For situation b), the inter-type method declarations may cause lookup impact, but we cover this with the analysis to deal with inter-type method declaration. Therefore, for the inter-type-implements-declaration itself, it can't cause any lookup impact.

7.5 Experimental Results

7.5.1 Examples

First, we explore our analyses with two examples: the *bank* example discussed in Section 3.2.1 and the *broken dispatch* example adapted from the program presented by Allen [All01].

Bank

A segment of our analyses report related to inter-type declarations in the bank example is shown in Listing 7.1. For presentation purposes, we omit the part related to lookup impact. As expected and discussed in Section 3.6, our analysis reports that both the inter-type method declaration **`GoldCard.payment(...)`** (Listing 3.9, line 9) and the inter-type parents declaration `ValueCard extends RewardCard` (line 7) cause lookup impact.

The report shows the inter-type method declaration **`GoldCard.bonus`** causes method lookup change of the method call [GoldCard].payment(...). In addition, the analyses report what the change is for different places where the method call may appear. For example, if the method call appears within the `bank.creditcard` package, [GoldCard].payment(...) matches to `RewardCard.payment(...)` before applying the inter-type method declaration and refers to `GoldCard.payment(...)` instead after applying the inter-type method

```
CreditCardAspect.aj:9,8-61 ITD: GoldCard.payment(bank.account.StudentCheckingAccount)
lookup impact:
   [bank.creditcard.GoldCard].payment(bank.account.StudentCheckingAccount)
      within declaring class, originally matched to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount), currently
          matches to bank.creditcard.GoldCard.payment(bank.account.StudentCheckingAccount)
      within declaring package, originally matched to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount), currently
          matches to bank.creditcard.GoldCard.payment(bank.account.StudentCheckingAccount)
      within class or subclasses declaring protected member, originally matched to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount), currently
          matches to bank.creditcard.GoldCard.payment(bank.account.StudentCheckingAccount)
      within other place, originally matched to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount), currently
          matches to bank.creditcard.GoldCard.payment(bank.account.StudentCheckingAccount)

CreditCardAspect.aj:7,1-47 ITD: ValueCard extends RewardCard
shadowing impact:
...
lookup impact:
   [bank.creditcard.GoldCard].payment(bank.account.AbstractAccount)
      within declaring class, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
      within declaring package, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
      within class or subclasses declaring protected member, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
      within other place, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
   [bank.creditcard.ValueCard].payment(bank.account.AbstractAccount)
      within declaring class, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
      within declaring package, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
      within class or subclasses declaring protected member, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
      within other place, originally matched to
          bank.creditcard.AbstractCreditCard.payment(bank.account.AbstractAccount),
          currently matches to
          bank.creditcard.RewardCard.payment(bank.account.AbstractAccount)
```

Listing 7.1 Report of analyzing inter-type declarations in bank example

declaration.

The report also shows that the inter-type parents declaration **ValueCard extends RewardCard** causes method lookup changes for both [GoldCard].payment(...) and [ValueCard].payment(...). Similarly, the analyses report what the change is for different places where the method call may appear.

Broken Dispatch

This example is originally presented by Allen [All01] and is a Java program. In this example, a `LinkedList` that is assumed to only contain `String` is implemented. First, a `List` interface is designed and specifies basic functions provided. Then, a `Cons` class implements `List` interface, and declares two fields: `Object first`, which contains the data of the current item; and `List rest`, which is a `List` contains the rest of the list. `Cons` provides two constructors: one constructor takes one `Object` type of argument and initializes `first` as the argument and `rest` as an empty list; another constructor takes two arguments and initialize the two fields as the two arguments. Finally, the `LinkedList` is implemented using the `Cons` data structure. The program works well and can pass all tests. Later, a new `Cons` constructor is added, which takes a `String` representation of list as argument, splits the string into tokens and treats each token as a data item. After adding this new constructor, some tests suddenly break. The reason is that the `LinkedList(String)` constructor calls the `Cons(String)` constructor, and the call matches to `Cons(String)` now, which tries to split the string and adds tokens into the list; but before adding the new constructor it matches to `Cons(Object)`, which treats the string as a whole. We adapted this program into an AspectJ program by changing the new constructor introduction into an inter-type constructor declaration and expect that our analysis reports that the constructor lookup changes after applying the new constructor.

Listing 7.2 shows the report of analyzing the broken dispatch example. As expected, our analyses report that the inter-type constructor declaration causes lookup impact, and report [Cons](String), which originally matches to `Cons(Object)`, now matches to `Cons (String)`. Here, our analyses only reports two places — the declaring class and the declaring package — because the `Cons` class is package visible, so the constructor can not

be accessed outside the package; thus no lookup impact would be caused in protected and other invocation places.

```
ConsAspect.aj:3,1-15:2 ITD: Cons(java.lang.String)
lookup impact:
   [Cons](java.lang.String)
      within declaring class, originally matched to
         Cons(java.lang.Object), currently matches to
         Cons(java.lang.String)
      within declaring package, originally matched to
         Cons(java.lang.Object), currently matches to
         Cons(java.lang.String)
```

Listing 7.2 Report of analyzing the source code repository example

7.6 Benchmarks

Table 7.2 shows statistics of analyzing these benchmarks. The second column "C&A" shows the total number of classes and aspects. The third column shows the total number of inter-type declarations. The next column shows the number of lookup impacts. The last column shows the number of classes and aspects checked when discovering these lookup impacts manually. Similar to what we did for shadowing impact, we drew the partial UML graph[2], and then examined the hierarchy of the UML graph to determine if there is lookup impact caused. During this process, we recorded the number of aspects and classes visited.

From Table 7.2, we can find that for most benchmarks, when discovering these lookup impacts manually, at least 70% of classes and aspects need to be checked. For ProdLine, we have to check all classes and aspects since inter-type declarations are heavily used in it. Therefore, we can image the difficulty of classifying and understanding computation impacts in a large-scale application. However, by using our tool, it only takes few minutes to get complete lookup impact information.

When discovering these lookup impacts manually, we found that these lookup impacts are caused by overridden method. We believe the reason is that lookup impacts not caused

[2] The difference is here only methods members are concerned

benchmark	C&A	inter-type declarations	lookup	C&A manually
aopbank	6	1	0	3
bean	3	7	0	2
DCM	34	2	27	28
exptree6	12	27	4	11
observer	8	11	0	6
ProdLine	20	79	16	20
Tetris	17	0	0	8
tracing	6	0	0	2

Table 7.2 Basic statistics about benchmarks

by overridden methods usually leads to bugs, and as we mentioned when discussing benchmarks result about shadowing impact, these benchmarks are well tested; therefore, we could not find lookup impacts not caused by overridden method.

In addition, for the tracing benchmark, we state that the aspects are pure observers when introducing it in Section 2.2. From the above table, we can find that there is no lookup impact caused. Combining the result of state, computation and shadowing impact presented in Section 4.3.2, 5.3.2, 6.3.2 respectively, our tools verifies that aspects in tracing are pure observers since they change nothing in the rest of the program except adding computations.

Chapter 8

Visualization - Eclipse Plug-in

After all different kinds of impacts are analyzed and discovered, in addition to a text report regarding the analysis results, we integrated the analysis in an Integrated Development Environment(**IDE**) so that a programmer does not need to switch from the development environment to command line back and forth. Moreover, the GUI interface can give a better presentation of the analysis results than the text report. Since "integrating with existing UIs was more important than creating new ones" [KCCC06] and Eclipse is a existing multi-purpose development framework which includes tools for AspectJ development in an extensible graphical environment, we integrated **AIA** into Eclipse by developing an Eclipse plug-in.

8.1 Overview

Under the Eclipse platform, AJDT is the most popular IDE for AspectJ; thus our plug-in utilizes AJDT as the AspectJ IDE[1]. Figure 8.1 shows a snapshot of **AIA** Eclipse plug-in, which extends *menu* to run **AIA** (the part marked as 1), creates a new *view* to display the analysis result (the part marked as 2), and generates new *marker*s in editor to mark where impacts are caused (the part marked as 3).

[1] Another important reason is that abc does not have an Eclipse plug-in.

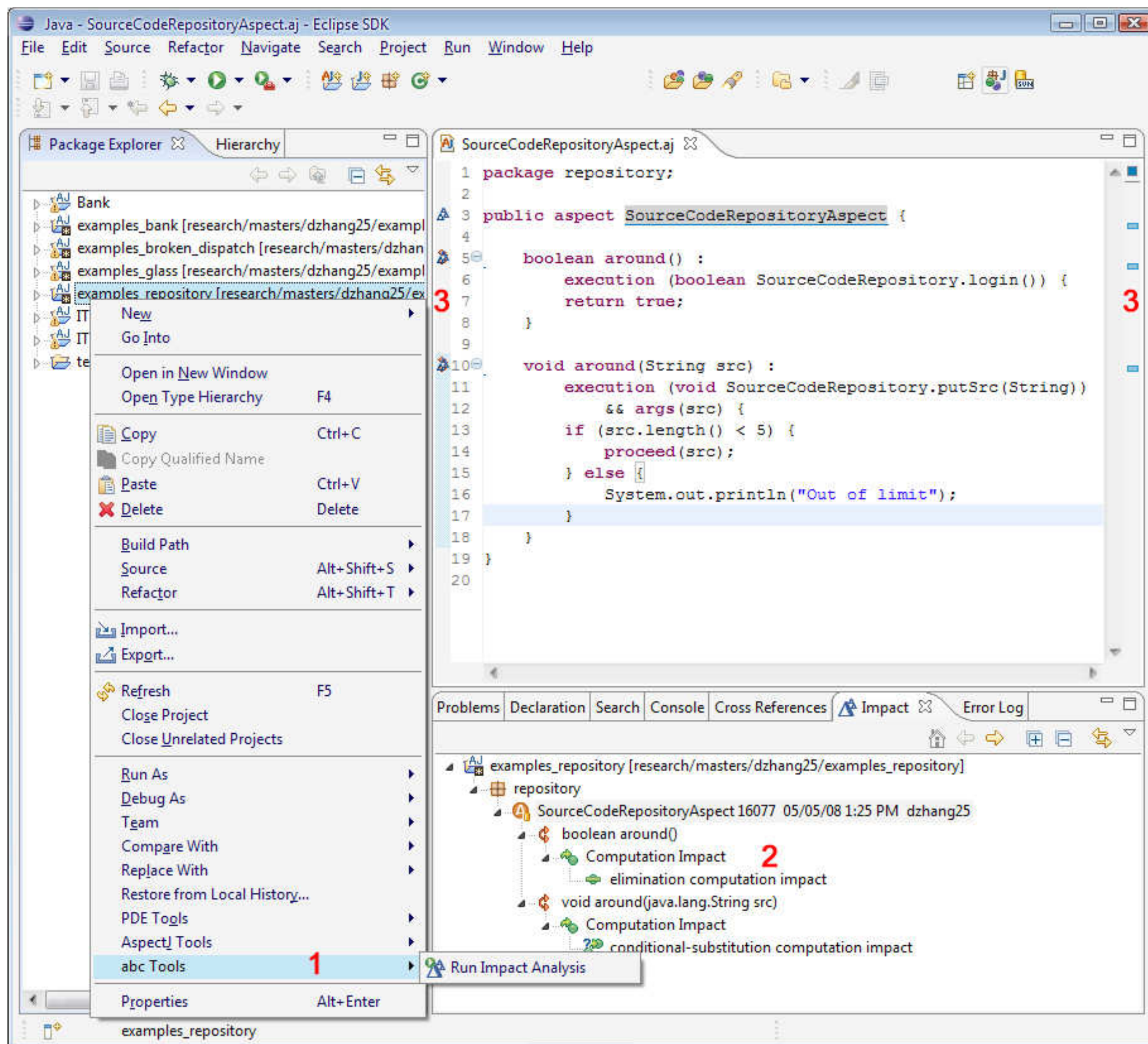

Figure 8.1 Eclipse plug-in snapshot

8.2 Running AIA

First we extends the pop-up menu to let the programmer run **AIA**[2].

When the user right clicks on a project, if the project is an AspectJ project, a new menu group called “abc Tools” will be displayed, and there is a menu called “Run Impact

[2]Since we use points-to analysis, which requires a certain amount of memory, we suggest to run Eclipse with “-vmargs -Xmx512m” argument to increase the heap size allocated to JVM.

Analysis" in this group (as shown in the part marked 1 in Figure 8.1); clicking this menu runs **AIA**. Since **AIA** uses points-to analysis, it needs the main class (the class containing the main() method) information. If there is more than one main class, a main class selection dialog will be displayed and list all main classes, as shown in Figure 8.2. The user can select a main class based on his/her need. By default, **AIA** is running in the foreground, as shown in Figure 8.3, but clicking the "Run in Background" button can run it in the background.

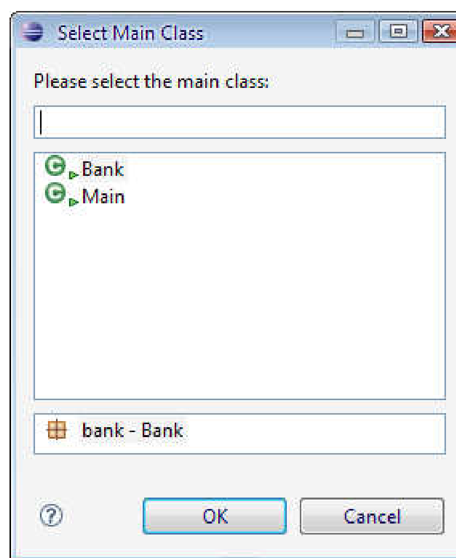

Figure 8.2 Main class selection dialog

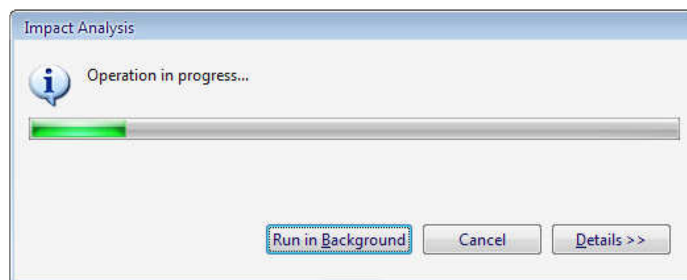

Figure 8.3 Running progress dialog

We implement the menu by extending the `org.eclipse.ui.popupMenus` extension point. We set the `projectNature` filter as `org.eclipse.ajdt.ui.ajnature`, thus this menu is displayed only when the selected item is an AspectJ project.

After the user clicks this menu, we first get the selected project and then fetch out all source files included in this project by querying AJDT. After that, main classes are searched from all source files using `MainMethodSearchEngine` of the Java Development Tool (**`JDT`**). If more than one main class is found, a standard `MainTypeDialog` will be created and displayed so that the user can specify the main class.

Then, we call abc with **AIA** extension enabled in a separate thread. After the analysis is done, we open the view or notify the view to refresh its content if it is already opened.

8.3 Impact View

Analysis results are displayed in a view called "Impact", as shown in the part marked 2 in Figure 8.1. Information in the view is displayed as a tree structure.

The top level displays the name of the project being analyzed. Double clicking the project name expands/collapses the tree.

Below the project name, we list packages. Double clicking a package expands/collapses the sub-tree of this package. Only those packages that contain advice or inter-type declarations causing impacts are displayed.

The next level lists Aspects in each package. Double clicking an Aspect opens an editor displaying the source code of this aspect. Only those aspects containing advice or inter-type declarations causing impact are displayed.

All advice and inter-type declarations that cause impacts are listed in third level. Double clicking an advice or inter-type declaration opens an editor highlighting the code corresponding to the advice or inter-type declaration, as shown in Figure 8.4.

Impacts caused are listed as children of advice or inter-type declarations.

For an advice, it may cause state impacts and computation impacts. For a computation impact, we display its name. For a state impact, the position of the code in the advice causes the impact is listed. Double clicking the position opens the editor containing the advice and highlighting the code causing the state impact. For a direct state impact, we display the field changed by this impact, and we display both the run-time type being changed and the class declaring the field (the run-time type may inherit this field from its super class). Double clicking the run-time type opens an editor containing its source file;

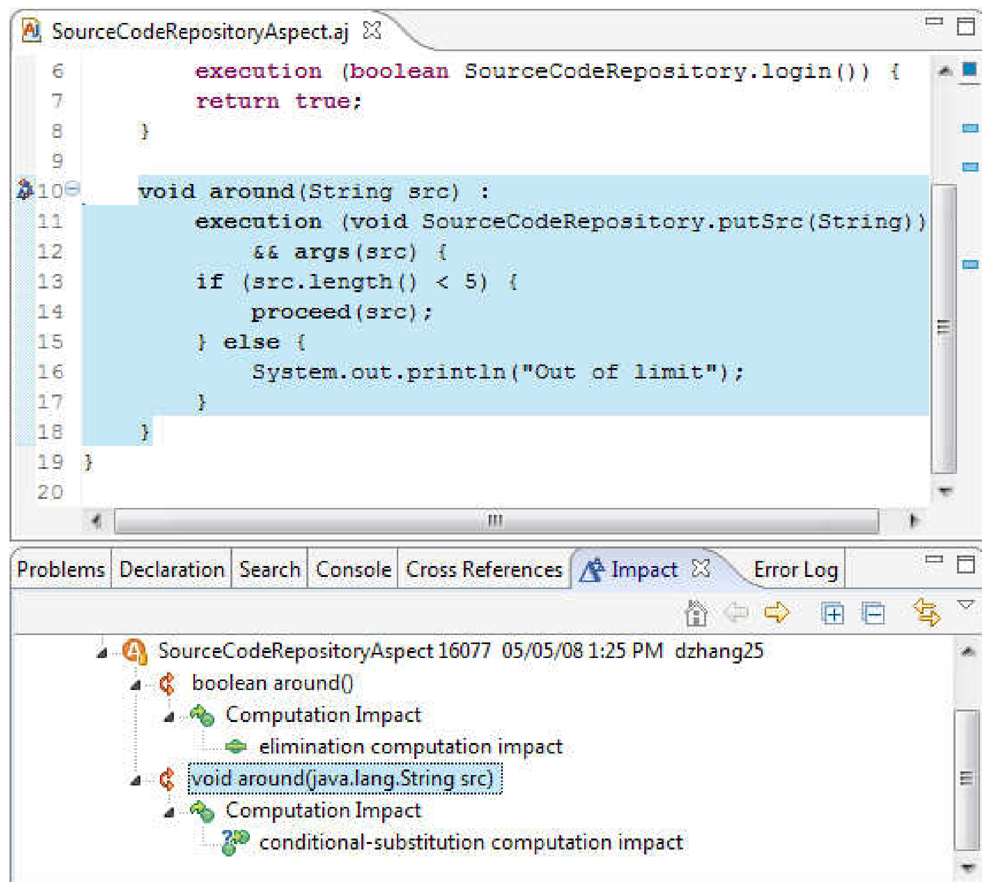

Figure 8.4 Advice highlighting

double clicking the declaring class opens an editor containing the class' source file and highlighting the field's declaration. For an indirect state impact, we list all actual positions (*i.e.*, position of evidences) causing field changes; double clicking a position opens an editor highlighting the code corresponding to the position. For each evidence position, we display the changed field information similar to the field involved in direct state impact, and double clicking works in the same way.

Figure 8.5 shows how the computation impact, direct and indirect state impact are presented, and shows the evidence highlighting (for presentation purpose, we add line numbers to the left of the view). The project being analyzed is the bank example presented in Section 3.2.1.

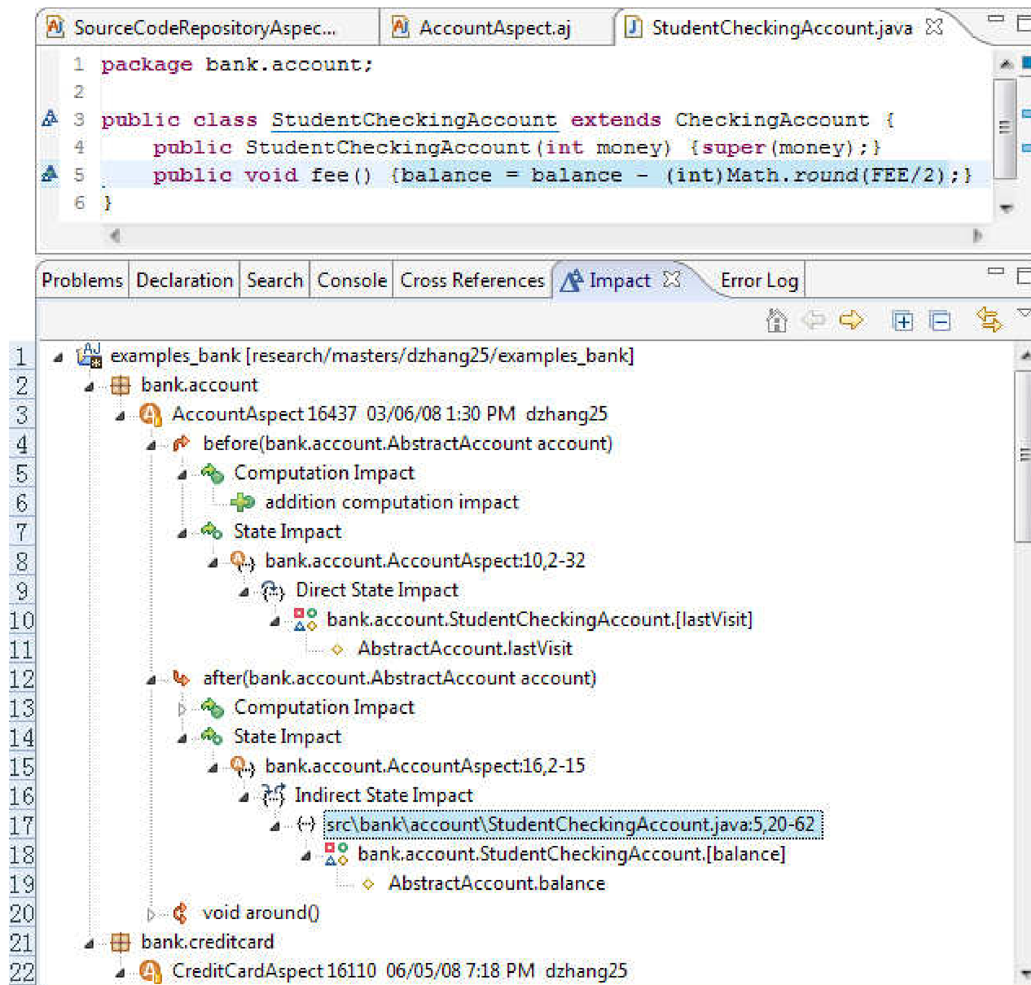

Figure 8.5 Impact view displaying computation and state impacts

First, we can see how the tree structure is organized. The first level (line 1) shows the project name. The second level shows packages, *e.g.*, the `bank.account` (line 2) and `bank.creditcard` (line 21) package. The third level shows aspects causing impacts, *e.g.*, the `AccountAspect` (line 3) and `CreditCardAspect` (line 22). The next level shows advice, *e.g.*, the `before(AbstractAccount)` advice at line 4, the `after(Abstract-Account)` advice at line 12 and the `void around()` advice at line 20.

The level below advice lists state impacts and computation impacts. An example of computation impact can be found at line 5-6, which shows the `before(AbstractAccount)`

advice causes “addition computation impact” (line 6). An example of direct state impact can be found at line 7-11. The advice `before(AbstractAccount)` causes state impact (line 7), and the position of the code causes the state impact is shown at line 8, which causes a “Direct State Impact” (line 9). The state impact changes the `lastVisit` field in class `StudentCheckingAccount` (line 10), and the field is declared in class `AbstractAccount` (line 11). An example of indirect state impact can be found at line 14-19. In addition to that the position of advice’s code causing indirect state impact is shown at line 15, the position of the code actually causes the indirect state impact (*i.e.*, evidence) is listed at line 17. The top part in Figure 8.5 shows the editor containing the source file listed at line 17 and highlighting the source code of the evidence after the user double clicks the evidence.

For an inter-type declaration, it may cause shadowing impacts and lookup impacts. For shadowing impact, we list all [affected type].field, and for each of them, we list the original type and current type. Double clicking [affected type].field opens an editor containing the affected type, and double clicking the original type and current type opens an editor highlighting the declaration of the field in the original or current type. For lookup impact, we list all [affected type].method, and for each of them, for different invocation place, we list the original method and the current method. Double clicking an [affected type].method opens an editor containing the affected type; double clicking an invocation place expands/collapse the sub-tree; double clicking the original method and current method opens an editor highlighting the method declaration.

Figure 8.6 shows how the shadowing and lookup impacts are presented (for presentation purpose, we add line numbers to the left of the view). Same as advice, inter-type declarations are listed below the level of aspects, *e.g.*,, `declare parents` at line 3, `GoldCard.bonus` at line 6 and `GoldCard.payment(...)` at line 11. The level below inter-type declarations lists shadowing and lookup impacts. An example of shadowing impact can be found at line 7-10. Line 8 shows that a field reference of `bonus` field of affected type `GoldCard` is affected; the original type is `RewardCard` (line 9), and the current type is `GoldCard` (line 10). An example of lookup impact can be found at line 12-25. Line 13 shows that a method call of `payment(...)` on the receiver of type `GoldCard` is affected. Then, in the next level, different invocation places are listed, *e.g.*, *class* at line 14,

package at line 17, *protected* at line 20 and *other* at line 23. Below each invocation place, the first and second line show the original and current method respectively. For example, line 15 shows the original method is `RewardCard.payment(...)` in the declaring class, and line 16 shows the current method is `GoldCard.payment(...)` in the declaring class.

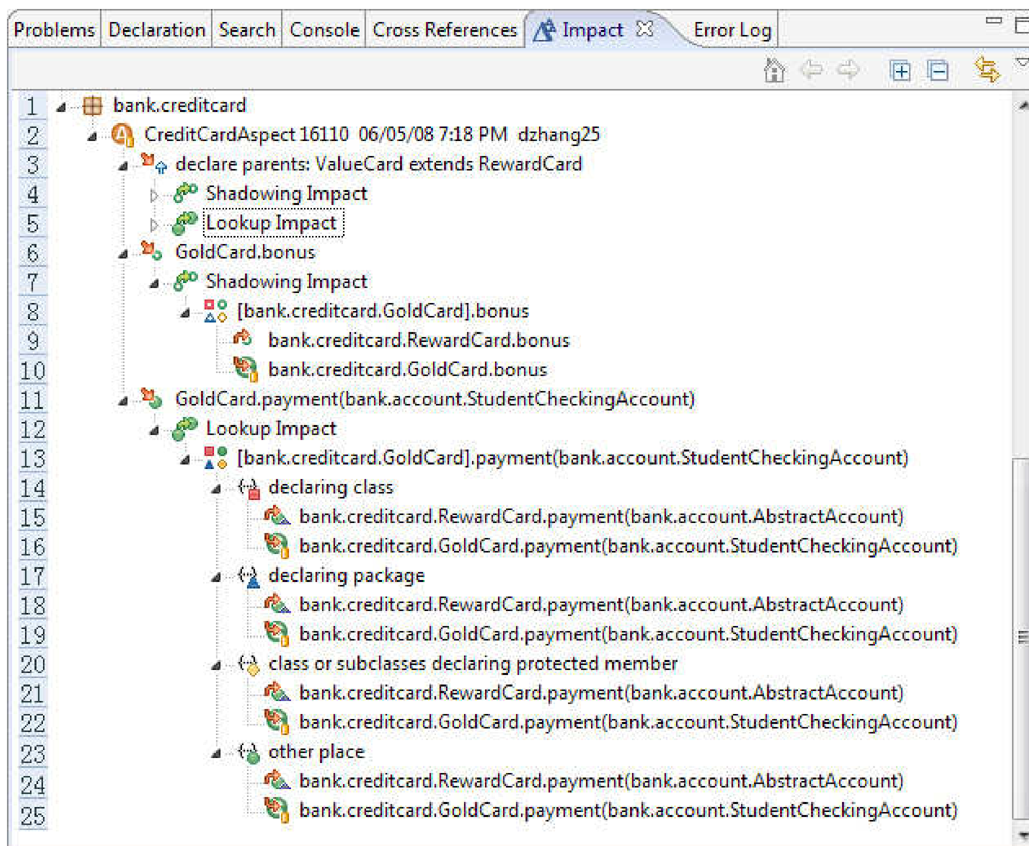

Figure 8.6 Impact view displaying shadowing and lookup impacts

On the top of the view, there are six buttons. The first three are standard Eclipse tree view navigation button. The fourth (showing a "+" sign) and fifth (showing a "-" sign) button are the button to expand and collapse the whole tree respectively. The last button is the button to enable/disable the "link with editor" function. The "link with editor" function allows the content of the view changing according to the cursor position in the editor. As shown in Figure 8.7, the "link with editor" function is enabled. After clicking the line 10 of AccountAspect.aj, the view shows only the impact information relating to this line, *i.e.*, the

computation impact caused by the advice and the state impact caused by this line of code.

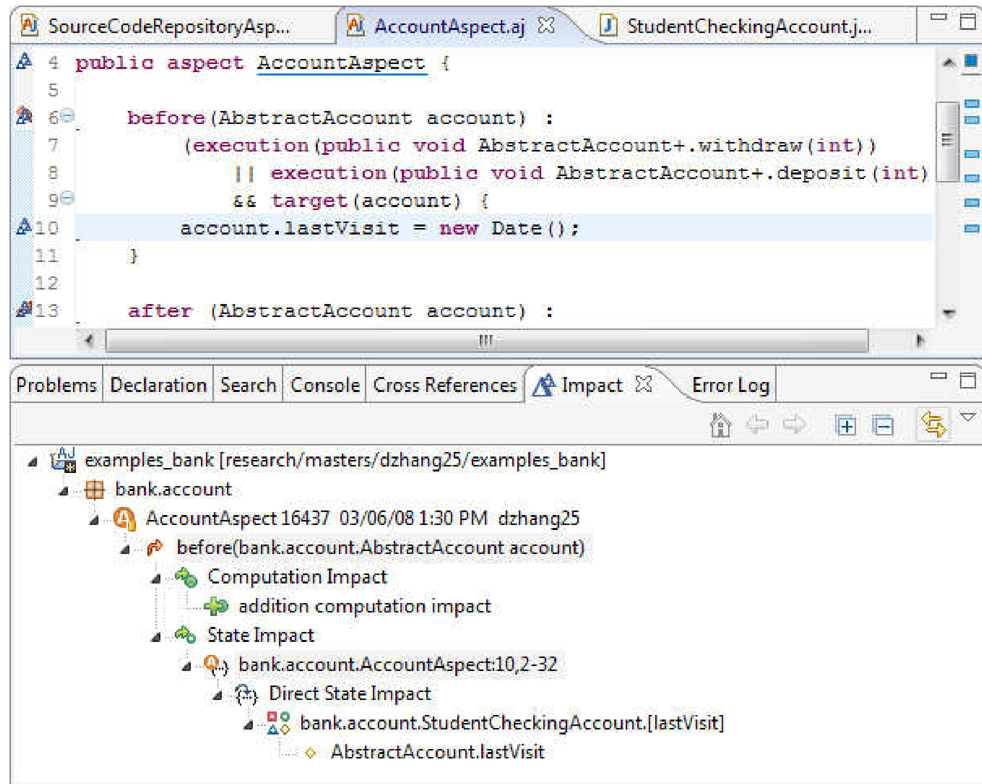

Figure 8.7 Impact view linking with the editor

By double clicking an item in the view, a user can navigate from impact to the cause of the impact, and by enabling "link with editor " function, a user can navigate from a cause of impact to impacts caused. Therefore, the user gets a two-way navigation of the information, and the ability of the user reasoning the impact is greatly improved.

The view is implemented by extending the `org.eclipse.ui.views` extension point. Inside the view, we create a JFace `TreeViewer`. After the analyses are done, we create a tree model and feed this model to the `ImpactViewContentProvider` (created by implementing JFace `IStructuredContentProvider` and `ITreeContentProvider`) of the `TreeViewer`. We first design the `AbstractTreeNode` which defines the basic functionality of a tree node, such as providing the name and image to the `ImpactViewLabelProvider` (extending JFace `LabelProvider`), navigating to parent/childern. We then design differ-

ent kinds of concrete `TreeNode` to represent different items in the view, and each concrete `TreeNode` defines its own label and action when double clicking. Moreover, we create the `ImpactViewSorter` (extending JFace `ViewerSorter`) to determine the display order of tree nodes in the same level.

When implementing the "link with editor" function, we first get the current position of the cursor in the active editor, and then traverse the tree model to select items related to the cursor position by comparing the position information contained in the `TreeNode` to the cursor position. After that, we feed the selected nodes to the `ImpactViewFilter` (extending JFace `ViewFilter`), which refreshes the content of the view.

8.4 Impact Marker

In addition to display impact in the view, we add markers to editor, as shown in the part marked 3 in Figure 8.1. Code involving in impact is marked by makers both on the vertical ruler (on the left side of the editor) and the overview ruler (on the right side of the editor). Figure 8.7 shows markers on AccountAspect.aj. These markers also ease the use of "link with editor" function as they works like hotspots to remind the user the position of code involving in impact. In case that the position is not an entire line, we underline the actual code involving in impact, for example, the "AccountAspect" at line 4 in Figure 8.7.

We extend the `org.eclipse.core.resources.markers` extension point to create our own marker type, and our marker type extends `org.eclipse.core.resources.textmarker`. To define the appearance, *i.e.*, icon, color and style, of our marker type, we extends the `org.eclipse.ui.editors.annotationTypes` and `org.eclipse.ui.editors.markerAnnotationSpecification` extension points. Markers are created after the tree model is generated. We traverse the tree model and create corresponding markers by querying position information contained by tree nodes.

8.5 Summary

With the extended menu, we allow **AIA** to be run in GUI in addition to command line. With the interactive view, we not only give a graphical presentation of impacts but also

allow programmers to interact with impacts. With the markers, we allow programmers to be aware of impacts even when browsing source code. With the “link with editor” function, we create a linkage between the view and editor, thus allowing programmers to navigate impacts information in a two-way manner. Therefore, with this Eclipse plug-in, we have integrated our analysis into the AspectJ IDE, rather than just extending it.

Chapter 9
Related Work

The issue of helping the programmer understand the impacts of aspects is not a new one and there has been some interesting previous work in this area. Some of them tried to analyze, categorize and classify aspects; some of them tried to improve the AOP language and enhance reasoning about the AO software.

9.1 Analyzing, Categorizing and Classifying Aspects

Störzer noticed the lookup of method calls might change due to static-crosscutting [SK03]. Changes caused by AspectJ introduction (*i.e.*, inter-type method and constructor declaration) and direct hierarchy modification (*i.e.*, inter-type parents declaration) were defined based on semantical changes in the hierarchy, and were analyzed using the interference criterion introduced by Snelting [ST02]. However, only a prototype of the analysis was implemented due to the limitation of the hierarchy analysis. After that, Störzer stated that advice may change the control-flow and state of base programs [Stö03a]. Although he mentioned that aspect analysis (also for the lookup change mentioned in [SK03]) requires data flow analysis, but he had no infrastructure tools available, thus he later proposed using trace analysis to fulfill the impact analysis. His approach relies on comparison of two traces of the program without and with aspect applied for a single test. Then, by identifying patterns of differences, the impact of an aspect can be observed [SKB03]. However, first, his approach heavily depends on the quality of the test case; second, the report can only

vaguely describe the impact at the level of aspect. Since our infrastructure does support data flow analysis, we were able to actually implement static impact checking.

The work from MIT by Rinard et. al. [RSB04] is very related, which was also designed to work on AspectJ and used static analysis. They did not mention the change caused by static crosscutting, but there are clearly similarities between the two approaches when discovering impact caused by advice declarations as both seek to summarize state impacts and computation impacts (although these were not the terms used in the MIT paper). From a conceptual point of view the approaches differ in the manner in which the impacts are abstracted. For example, in the MIT approach, state impacts are expressed as interferences between fields accessed by a base program method and fields accessed by an advice, whereas our approach takes a more advice-centric approach, and we report all fields of the base program written by an advice. We believe that our approach will lead to more direct report and is more easily integrated into an IDE (as shown by our Eclipse plug-in). From an implementation point of view there are also similarities and differences. Both systems are built on Java bytecode frameworks which support points-to analysis. The MIT prototype used the bytecode produced by ajc as input, and used the MIT Flex compiler infrastructure for the static analyses. Their implementation was limited to method call and method execution join points, perhaps because of the loose coupling between ajc and Flex. Our approach is implemented directly in the abc compiler and so we have access to all the necessary information to handle all kinds of join points. Further, our implementation includes analysis regarding static crosscutting and the visualization of analysis result.

[Kat06] categorizes aspects by identifying temporal properties using temporal logic. Semantics of an aspect system is expressed using UML statecharts, and the weaving process is considered as a transformation from the original state graph to an augmented one. Based on how the temporal properties are changed, three categories of aspects are defined. *Spectative aspects* can't modify any variables or change the flow of control of the base system. *Regulative aspects* can change the flow of control of the base system by "restricting operations, delaying some operations, or preventing the continuation of a computation". *Invasive aspects* also modify variables in the base system. For each category of aspects, an outline of static analysis to syntactically identify if an aspect belongs to that category is presented. Their base system means a system before any aspects are woven into it, which is

not as straightforward as our definition from a programmer's point of view and does cause the analysis determining aspects interference to be very complicated. Their categorizes of aspects can be expressed by a combination of our state impact and computation impact. Spectative aspects have no state and computation impact; regulative aspect have computation impact but no state impact; and invasive aspects have both state and computation impact. Moreover, our classification further categories the effect of changing flow of control of the base system so that programmer can get more detailed information. In addition, only an outline of static analysis was presented, and no implementation was provided.

9.2 Improving AOP Language and Enhancing Reasoning

Starting at the syntactic/IDE level, the eclipse plug-in AJDT[1] [AJD07] provides visualizations to indicate shadows where advice applies in the base program, thus providing some cues for the programmer as to places in the base program which might be affected. However, to discover the actual impacts of aspects on the base program, a programmer has to manually review the source code and possibly has to frequently switch between base program source and aspect source. A key difference in our approach is that we are using static analysis to find more detailed information for programmers. We leverage the visualization tools in AJDT by exposing our more detailed information to programmers through our Eclipse plug-in.

Dantas introduced the concept of *harmless advice* [DW06], which works like ordinary aspect-oriented advice but is designed to obey a weak non-interference property, *i.e.*, it may change the base program's termination behavior and use I/O, but it does not influence the final result of the mainline program. In order to detect and enforce harmlessness, they defined a novel type and effect system related to information-flow type systems. They also presented an implementation of the language. However, their work was done at a very abstract level and is hard to integrate it directly into AspectJ. Our approach is more focused on developing classifications and associated analyses that have been integrated into

[1] http://www.eclipse.org/ajdt/

an AspectJ compiler.

Recebli analyzed different ways aspects can break encapsulation and proposed the purity aspect language feature to AspectJ [Rec05]. Through this feature, a programmer can declare an aspect is *pure on* a specified set of classes by promising that the aspect will not change the behavior of the set of classes. Moreover, he presented an implementation of the proposed purity annotation as an abc extension and used static analysis to verify purity. Our approach is more focused on the non-pure impacts and on ways of categorizing and approximating those impacts. Our static analysis is also more detailed as we also take into consideration side effects of method calls when analyzing **`proceed`**.

Aldrich presented *Open Modules* [Ald05], which focused on modular reasoning about advice and is claimed by the author as “a new module system for languages with advice”. In Open Modules, pointcuts are considered as part of interface exposed by a module. Therefore, both pointcuts and traditional interfaces may be exposed to client, but the implementation detail is hidden. External advice can only advise internal event through pointcuts exposed by a module. The maintainer of a module is responsible for maintaining “the semantics of exposed pointcuts as the module’s implementation evolves”. To show Open Modules ensures that a module’s semantics is preserved when changing its implementation so that programmers can reason about advices inside a module, a formal model of advice, TinyAspect, was presented, Open Modules is added into the model, and the oberservational equivalence of Open Modules is proved under this formal model. Our approach helps programmer reasoning about advice by concluding impact caused by applying them to the base program without extending the aspect oriented language. In addition, our classification and analysis are still valid in Open Modules since although it ensures the local reasoning about advice, the impact caused by advice still need to be classified and concluded by static analysis, and Open Modules may reduce the cost of static analysis by ensuring local reasoning.

Cliftion presented *Modular Aspects with Ownership* [CLN07], MAO,

which was designed as a variant of AspectJ 5 by introducing new annotations and utilizing generic aspect technique. The purpose of MAO is to simplify reasoning “whether one module (class, method, aspect, advice) may potentially affect the behavior of another module”. MAO achieves this by allowing programmers to state restrictions on control and heap effects of advice by adding annotations. A control effect is caused by perturbating the

program's flow of control. A heap effect is caused by assigning some object fields. MAO modularly checks the validation of control-limited of an advice using simple desugaring and conservative criteria. A heap effect is identified utilizing the concern domains, which "is an ownership type-and-effect system" and requires programmers to identify and annotate objects with a particular owner, *i.e.*, concern domain. Concern domains are declared explicitly by programmers. The heap effects of method and advice and concern domains accessed by classes or aspects need to be specified with annotation by programmer, and MAO validates the access declaration of classes or aspects by checking only annotations. In addition, a new pointcut designator called writes is introduced, which has the form writes(D) and "matches all join points that may write to concern domain D". Heap and control effect in MAO are similar to our state and computation impacts, but we further distinguish different kinds of state and computation impacts. Moreover, effects in MAO are identified and annotated by programmer manually, and we use static analysis to identify these impacts automatically. In addition, MAO only takes care of exceptions when programmers declare control limited advice and leaves other considerations to programmers.

Chapter 10
Conclusions and Future Work

10.1 Conclusions

As AOP, especially AspectJ, is becoming increasing popular, we believe that tools can help programmers understand the complicated interaction between aspects and base programs. The design of tools that can compute both useful and accurate information presents many interesting and challenging problems, and the availability of such tools should help increase the adoption of AOP.

We presented different ways that advice and inter-type declarations can interfere with the state and computation of a base program and proposed a concise classification of impacts caused by them crosscutting the base program. We classified these impacts as *state impacts* — which are caused by advice affecting the state of base program, and *computation impacts* — which are caused by advice affecting the computation of base program, *shadowing impacts* — which are caused by inter-type declarations affecting the field reference of base program, and *lookup impacts* — which are caused by inter-type declarations affecting the method lookup of base program. In addition, we further classified *state impacts* into *direct state impacts* and *indirect state impacts*; and classified *computation impacts* as *addition*, *elimination*, *definite-substitution*, *conditional-substitution* and *mixed*. Our definition of a base program follows the principle of “everything except me”, and our tool covers interference between aspects naturally.

Based on this classification, we implemented static impact analyses in the abc compiler to analyze all kinds of advice and inter-type declarations. By using the points-to analysis and side-effect analysis supported by Soot, our impact analyses system can give precise estimations of impacts. Our approach also produces an informative analysis report. In the report, we not only report the impact information, but also report the causes of impacts, so we can guide the programmer to understand the key impacts of aspects on their program.

We also integrated these analyses into an IDE. We integrated our toolkit into Eclipse and produced an interactive tree-structure view of the report that both allows programmers to view the impacts in a more graphical manner and allows programmers to navigate directly to the parts of the source code involved in the impacts. In addition, we generate markers into the source code editor so that programmers can be aware of the impacts when browsing the source code. Moreover, we provide the function to link the impact report view to the current editing location in the source code editor so that programmers can focus on the impacts only related to the source code being edited.

10.2 Future Work

There are also several extensions to the analyses that we would like to undertake.

First, it would be interesting to extending the analysis to cover programmer-specified unchecked exceptions when discovering *exact-proceed*s — *i.e.*, a programmer could specify a set of unchecked exceptions, and the `ExceptionBeforeProceedAnalysis` would analyze if these exceptions may thrown before the **`proceed`** call — because in practice, unchecked exceptions are used more frequently than checked exceptions.

We would also like to experiment with different points-to analyses, since the precision of our impact analyses heavily depends on the points-to result. There are two interesting context-sensitive analyses available for Soot now, the Paddle framework [Lho06], and the demand-driven analysis of Sridharan and Bodik [SB06]. It should be simple to integrate these into our approach, and a study of the effect of points-to precision on the quality of impact reports would be very interesting.

Finally, an extension that analyzes and reports the shadowing impact on the actual reference site of field reference would make the toolkits more complete. Adding the actual

reference site analysis on top of our analyses is very straightforward, and the basic idea is simple: for each field reference in the application, check if the receiver's type and the name of field match to one in our shadowing impact set; if there is a match, report the original type and current type discovered by our shadowing impact analysis.

Bibliography

[ACH+05] 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*, Chicago, Illinois, 2005, pages 87–98. ACM, New York, NY, USA.

[AJD07] AJDT team. AspectJ Development Tools (AJDT). http://www.eclipse.org/ajdt, 2007.

[Ald05] Jonathan Aldrich. Open Modules: Modular Reasoning About Advice. In Andrew P. Black, editor, *ECOOP*, 2005, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer.

[All01] Eric Allen. Diagnosing Java Code: The Broken Dispatch bug pattern. http://www.ibm.com/developerworks/java/library/j-diag7.html, May 2001.

[BLQ+03] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, USA, 2003, pages 103–114. ACM Press.

[Bra03] Ryan Brase. Avoid these Java inheritance gotchas. http://articles.techrepublic.com.com/5100-22-5031837.html, June 2003.

[CLN07] Curtis Clifton, Gary T. Leavens, and James Noble. MAO: Ownership and Effects for More Effective Reasoning About Aspects. In Erik Ernst, editor, *ECOOP*, 2007, volume 4609 of *Lecture Notes in Computer Science*, pages 451–475. Springer.

[dM04] Oege de Moor. abc: an Implementation of AspectJ. Seminar at the Computer Laboratory, Cambridge, United Kingdom, http://abc.comlab.ox.ac.uk/documents/dec8.pdf, December 2004.

[Duc06] Allison Duck. Implementation of AOP in non-academic projects. In *AOSD'06 - Industry Track Proceedings*, March 2006, pages 68–77.

[DW06] Daniel S. Dantas and David Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Charleston, South Carolina, USA, 2006, pages 383–396. ACM Press, New York, NY, USA.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, Orlando, Florida, United States, 1994, pages 242–256. ACM Press, New York, NY, USA.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall PTR, 2005.

[Inc03] Object Technology International Inc. Eclipse Platform Technical Overview. http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.

[Kat06] Shmuel Katz. Aspect Categories and Classes of Temporal Properties. *Transactions on Aspect Oriented Software Development (TAOSD)*, pages 106–134, 2006. LNCS 3880.

[KCCC06] Mik Kersten, Matt Chapman, Andy Clement, and Adrian Colyer. Lessons learned building tool support for AspectJ. In *AOSD'06 - Industry Track Proceedings*, March 2006, pages 49–57.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, April 2003, volume 2622 of *LNCS*, pages 153–169. Springer, Warsaw, Poland.

[Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[Lho06] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[Raz99] Chrislain Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, December 1999.

[Rec05] Elçin A. Recebli. Pure Aspects. Master's thesis, Oxford University, September 2005.

[RSB04] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, Newport Beach, CA, USA, 2004, pages 147–158. ACM Press, New York, NY, USA.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, Ottawa, Ontario, Canada, 2006, pages 387–400. ACM Press, New York, NY, USA.

[SHR+00] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, 2000, pages 264–280.

[SK03] Maximilian Störzer and Jens Krinke. Interference Analysis for AspectJ. In *Foundations of Aspect-Oriented Languages Workshop*, 2003.

[SKB03] Maximilian Störzer, Jens Krinke, and Silvia Breu. Trace Analysis for Aspect Application. In *Analysis of Aspect-Oriented Software (AAOS)*, 2003.

[ST02] Gregor Snelting and Frank Tip. Semantics-Based Composition of Class Hierarchies. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002, pages 562–584. Springer-Verlag, London, UK.

[Stö03a] Maximilian Störzer. Analysis of AspectJ Programs. In *Proceedings of 3rd German Workshop on Aspect-Oriented Software Development*, 2003.

[Stö03b] Maximilian Störzer. Analytical problems and AspectJ. AOSD workshop, 2003. Talk, also available at http://www.infosun.fim.uni-passau.de/st/staff/stoerzer/stoerzer2003boston.pdf.

[SW07] Martin Sulzmann and Meng Wang. Aspect-oriented programming with type classes. In *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, Vancouver, British Columbia, Canada, 2007, pages 65–74. ACM Press, New York, NY, USA.

[VR00] Raja Vallé-Rai. Soot: A Java Bytecode Optimization Framework. Master's thesis, McGill University, July 2000.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, 1999, pages 125–135.