PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

**Migrating Object Oriented code to Aspect Oriented Programming**

Mariano Ceccato

Advisor:

Prof. Paolo Tonella

Università degli Studi di Trento

December 2006
Abstract

Aspect Oriented Programming (AOP) is a new programming paradigm that offers a novel modularization unit for the crosscutting concerns. Functionalities originally spread across several modules and tangled with each other can be factored out into a single, separate unit, called an aspect. Although AOP was originally proposed for the development of new software, systems written using traditional modularization techniques may also benefit from the adoption of the more versatile decomposition offered by AOP, in terms of code understandability and evolvability.

The goal of this thesis is to investigate automated techniques that can be used to support the migration of existing Object Oriented Programming (OOP) code to AOP. To migrate an application to the new paradigm, a preliminary identification of the crosscutting concerns is required (aspect mining). Then refactoring is applied to transform the scattered concerns into aspects. The proposed methods have been assessed on case studies for a total of more than half a million lines of code. One of the proposed aspect mining method has also been compared with other state of the art methods. They all have been applied to a common case study and the results have been used to propose a brand new aspect mining technique, based on their combination.

Keywords
[Aspect Oriented Programming, Aspect Mining, Refactoring, Empirical Study]
# Contents

1 Introduction .................................................. 1
   1.1 Motivation .................................................. 1
   1.2 The Problem ................................................. 3
   1.3 Approach ...................................................... 4
   1.4 Contribution of the thesis ................................. 5
   1.5 Structure of the Thesis ...................................... 7

2 State of the Art ................................................. 9
   2.1 AOP .......................................................... 9
   2.2 Approaches .................................................. 11
      2.2.1 Aspect Oriented Development ......................... 11
      2.2.2 An AspectJ primer ....................................... 18
   2.3 Aspect Mining ................................................ 21
   2.4 Refactoring .................................................. 25

3 Aspect Mining ................................................ 29
   3.1 Dynamic aspect mining ..................................... 29
      3.1.1 Dynamic feature location .............................. 29
      3.1.2 Dynamic aspect mining technique .................... 34
      3.1.3 Clarifying example .................................... 37
      3.1.4 Tool support ........................................... 39
      3.1.5 Contribution to the state of the art ................ 40
3.2 Aspectizable interfaces ........................................... 41
  3.2.1 Aspectizable interfaces definition ...................... 41
  3.2.2 Aspectizable interfaces identification .................. 45
  3.2.3 Tool implementation ........................................ 47
  3.2.4 Contribution to the state of the art .................... 48

4 Refactoring .................................................. 51
  4.1 Aspectizable interfaces ...................................... 51
    4.1.1 Refactoring process description ....................... 51
    4.1.2 Tool support ............................................. 56
  4.2 Pointcut extraction ......................................... 58
    4.2.1 Extract Beginning/End of Method/Handler .............. 61
    4.2.2 Extract Before/After Call .............................. 63
    4.2.3 Extract Conditional ..................................... 65
    4.2.4 Pre Return .............................................. 66
    4.2.5 Extract Wrapper ......................................... 67
    4.2.6 Extract Exception Handling ............................ 69
    4.2.7 Pointcut Abstraction .................................... 70
    4.2.8 Iterative Refactoring Process .......................... 71
    4.2.9 Tool Implementation ..................................... 74
  4.3 Contribution to state of the art ......................... 76

5 Assessment .................................................. 79
  5.1 Assessment Subjects ........................................ 80
  5.2 Dynamic Aspect Mining ..................................... 84
  5.3 Aspect mining comparison .................................. 88
    5.3.1 The fan-in analysis experiment ....................... 88
    5.3.2 The identifier analysis experiment ................... 90
    5.3.3 Selected concerns ....................................... 93
    5.3.4 Limitations ............................................. 97
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.5 Complementarity</td>
<td>98</td>
</tr>
<tr>
<td>5.3.6 Definition of the combined techniques</td>
<td>100</td>
</tr>
<tr>
<td>5.3.7 Analysis indicators</td>
<td>101</td>
</tr>
<tr>
<td>5.3.8 Experimental results</td>
<td>102</td>
</tr>
<tr>
<td>5.4 Pointcut extraction</td>
<td>104</td>
</tr>
<tr>
<td>5.4.1 Refactoring</td>
<td>104</td>
</tr>
<tr>
<td>5.4.2 Pointcut Abstraction</td>
<td>107</td>
</tr>
<tr>
<td>5.4.3 Separation of Concerns</td>
<td>108</td>
</tr>
<tr>
<td>5.4.4 Performances</td>
<td>112</td>
</tr>
<tr>
<td>5.4.5 Size</td>
<td>112</td>
</tr>
<tr>
<td>5.4.6 Lessons Learned</td>
<td>113</td>
</tr>
<tr>
<td>5.5 Aspectizable interfaces</td>
<td>115</td>
</tr>
<tr>
<td>5.5.1 Aspectizable interfaces identification</td>
<td>115</td>
</tr>
<tr>
<td>5.5.2 Refactoring the aspectizable interfaces</td>
<td>119</td>
</tr>
<tr>
<td>5.5.3 The empirical study</td>
<td>122</td>
</tr>
<tr>
<td>5.5.4 Experimental hypotheses</td>
<td>123</td>
</tr>
<tr>
<td>5.5.5 Internal quality attributes</td>
<td>124</td>
</tr>
<tr>
<td>5.5.6 External quality attributes</td>
<td>127</td>
</tr>
<tr>
<td>5.5.7 Internal quality attribute values</td>
<td>130</td>
</tr>
<tr>
<td>5.5.8 External quality attribute values</td>
<td>136</td>
</tr>
<tr>
<td>5.5.9 Discussion</td>
<td>141</td>
</tr>
<tr>
<td>6 Conclusions and Future Works</td>
<td>143</td>
</tr>
<tr>
<td>6.1 Conclusions</td>
<td>143</td>
</tr>
<tr>
<td>6.2 Future work</td>
<td>148</td>
</tr>
</tbody>
</table>

Bibliography

A Abstract Refactoring Descriptions
List of Tables

3.1 Programming languages and their supported programming paradigms. 30
3.2 Execution traces for a use case with logging enabled (resp. disabled). 38
5.1 Java programs, case studies of the present assessment. 81
5.2 JHotDraw Results (use-case specific concepts). 84
5.3 JHotDraw Results (generic concepts). 86
5.4 Assessment for the Undo seed. 87
5.5 Summary of the results of the fan-in analysis experiment. 89
5.6 Selection of results of the identifier analysis experiment. 91
5.7 A selection of detected concerns in JHotDraw. 93
5.8 Concerns identified by either dynamic or fan-in analysis. 99
5.9 Recalled methods and precision before and after combination. 102
5.10 Enabling OO transformations applied to the four case studies. 105
5.11 Aspect refactorings applied to the four case studies. 106
5.12 Entities (types) and relationships (dependences) in the base code. 109
5.13 Base code size (LoC) before and after refactoring, for the classes subjected to refactoring. 113
5.14 Data set under analysis. 116
5.15 Precision and recall of the four aspect mining methods. 118
5.16 Features of the source code under analysis. The last three columns show the unique implemented interfaces, among them the aspectizable interfaces, and the number of implementations of the aspectizable interfaces. 120

5.17 Aspectizable interfaces found in the source code under analysis. The number of implementations is given in the last column. 121

5.18 Experimental design of the empirical study to measure maintenance and understanding times. 129

5.19 Internal quality metrics, divided into quartiles and computed either on the original or on the refactored code. 131

5.20 Chi-square thresholds and values. For statistical significance, Chi-square > Threshold is required. 133

5.21 Understanding time (UTIME) and total maintenance time (MTIME) in seconds, measured in each experimental session. Average (Avg) and variance (Var) are also shown. 138

5.22 Two-factor Anova for MTIME and UTIME. SS = Sum of Squares, df = degrees of freedom, MS = Mean sum of Squares. For statistical significance, F > F-crit is required. 139
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of aspect showing the mechanics of introductions and pointcuts/advice.</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>The concept lattice for Table 3.1.</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>The concept lattice for Table 3.1, with sparse labeling.</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Class diagram for the binary search tree application.</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Concept lattice for the binary search tree application.</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>Class diagram for a portion of the collection framework from the Java standard library.</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Refactoring: Move interface implementation to aspect.</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Source code transformation operated by the refactoring.</td>
<td>54</td>
</tr>
<tr>
<td>4.3</td>
<td>Abstraction of aspect computations into a super-aspect.</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>Refactoring toolkit.</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of refactoring: Extract End of Method.</td>
<td>62</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of refactoring: Extract After Call.</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>Example of refactoring: Extract Conditional.</td>
<td>65</td>
</tr>
<tr>
<td>4.8</td>
<td>Example of refactoring: Pre Return.</td>
<td>66</td>
</tr>
<tr>
<td>4.9</td>
<td>Example of refactoring: Extract Wrapper.</td>
<td>67</td>
</tr>
<tr>
<td>4.10</td>
<td>Example of refactoring: Extract Exception Handling.</td>
<td>68</td>
</tr>
<tr>
<td>4.11</td>
<td>Refactoring process overview.</td>
<td>69</td>
</tr>
<tr>
<td>4.12</td>
<td>Aspect selection dialog (a); Refactoring selection dialog (b); Preview of the resulting source code (c).</td>
<td>71</td>
</tr>
</tbody>
</table>
5.1 Example of merged advices from *JAccounting*. 108
5.2 Excerpt from *JHotDraw’s* class diagram. 111
5.3 Class diagram for the package *java.util*. Interfaces migrated to aspects are in bold boxes. 116
5.4 Class diagram for the package *java.util*, with interfaces migrated to aspects in bold boxes (a). At the bottom (b), class diagram (principal decomposition) after migration of the aspectizable interfaces. 134
5.5 Aspect diagram for the package *java.util*, representing the *Cloneable* (left) and *Serializable* (right) concern. 135
A.1 Mechanics for refactoring: Extract End of Method. 162
A.2 Mechanics for refactoring: Extract After Call. 166
A.3 Mechanics for refactoring: Extract Conditional. 167
A.4 Mechanics for refactoring: Pre Return. 168
A.5 Mechanics for refactoring: Extract Wrapper. 169
A.6 Mechanics for refactoring: Extract Exception Handling. 170
Chapter 1

Introduction

1.1 Motivation

In all engineering disciplines, humans cope with complex systems by using a “divide and conquer” approach: they divide a complex problem into many simpler sub-problems. Each sub-problem instance should have a low level of coupling with the other ones, so the team assigned to this sub-problem can reason about it as a whole and can devise a solution for it. The composition of the sub-problem solutions produces the solution for the whole complex problem.

In software engineering, many modularization mechanisms have been introduced. For example object oriented programming (OOP) provides the object model. This model usually fits a natural decomposition of the domain, because it tends to break systems down into units of data and behavior that correspond to real world entities. Whatever decomposition type is used, it becomes the main architecture of the system under development, and all future changes will refer to it.

Software systems are so complex that they cannot be developed without dividing them into sub-modules. The main drawback in this approach is that there are some system functionalities that cannot be assigned to a single module in the system decomposition. Examples of functionalities
that suffer this problem are persistence, error management and logging. Since the code fragments that implement these concerns are spread across many units, they are called *Crosscutting Concerns*. Crosscutting concerns violate the modularization goal that a system is decomposed into small independent parts: their main characteristic is that they are transversal with respect to the units in the principal decomposition, i.e., their implementation consists of a set of code fragments distributed over a number of units.

Reasoning about crosscutting concerns can be quite difficult, because it requires to deal with a lot of modules at the same time, in that there is no modularization support for them. For example, to deal with the persistence functionality, we have to understand all the pieces of code that perform persistent storage and retrieval. Crosscutting concerns are sources of problems, because their modification requires that:

- all code portions where such a functionality is implemented must be located (problem: scattering);

- all ripple effects associated with the changes must be determined (problem: tangling).

Aspect Oriented Programming (AOP) aims at solving the two main problems of crosscutting concerns, namely scattering and tangling, by providing a unique place where the related functionalities are implemented. A new modularization unit, called *aspect*, can be defined to factor out all code fragments related to a common functionality, otherwise spread all over the system. For example, an application can be developed according to its main logical decomposition, while the possibility to serialize and de-serialize some of its objects can be defined in a separate aspect.

Sometimes, the application being developed has a principal decomposition that can be completely decoupled from its aspects. Such a property
is called obliviousness. An application is oblivious of an aspect if it can be
developed independently of it, and the aspect can be added (or removed) later, by compiling (weaving) the application with/without it. Oblivious-
ness is not expected to hold for all aspects. For example, when persistence
is implemented as a separate aspect, the application can be oblivious of this functionality, except for the deletion and retrieval of objects, which
require two explicit invocations inside the modules in the principal decom-
position [52] (partial obliviousness).

A related issue is optionality. An aspect may either be an optional or an
integral part of an application. In the latter case, the application cannot
be compiled without the aspect. However, it is not required that aspects
be always optional features. In some cases, an application may require
some of its aspects to work properly.

1.2 The Problem

Aspect Oriented Programming (AOP) provides explicit constructs to de-
velop software systems whose crosscutting concerns are better modularized
because they are no longer tangled together and they are clearly separated
from the principal decomposition. In order to extend the benefits of AOSD
to already existing systems, a significant reverse and re–engineering effort
is required. The effort consists, first of all, of analysing the existing ap-
lication source code looking for those portions that implement the cross-
cutting functionality. The second part of the work is the transformation
of the existing program into an aspect-oriented reformulation. In order to
take advantage of the potential benefits of the AOP style of programming,
there is a need for migration support of existing applications and systems.
This dissertation presents a semi-automated approach to the process of
migration from the Object Oriented Programming paradigm (in Java) to
1.3. APPROACH

the Aspect Oriented Paradigm (in AspectJ).

Previous authors have identified two important phases in the migration from OOP to AOP: aspect mining [10, 48, 62, 66] and aspect extraction (including object-to-aspect refactoring) [33, 49, 68]. In aspect mining, the problem is to identify code which is likely to implement a crosscutting concern. This is a semi-automated process, in which a tool identifies candidate aspects for human consideration. As with other related tool-assisted code analysis techniques, there is a trade-off between the desire to locate all viable candidates, while identifying as few false positives as possible.

Software refactoring consists of the modification of internal program structure without altering the semantics (i.e., external behavior). It aims at improving internal quality factors (e.g., modularity), in order to make the code easier to understand and evolve. As with other migrations and conversions [15, 44, 69], the maintenance–migration effort involved in OOP to AOP must be automated to avoid un-sustainable cost.

In this thesis, the terms “migration” and “refactoring” have specific meanings and they are not interchangeable. “Migration” is used to indicate the complete process that changes an existing non-aspect (e.g. object oriented) system into an equivalent one that takes advantage of AOP to modularize crosscutting concerns. The term “refactoring” describes an activity that is performed many times during the migration process: it indicates a small, controlled and automated source code transformation. Refactoring consists of semantic preserving code transformations that improve internal code organization.

1.3 Approach

In this thesis, the problem of migrating an existing system from OOP to AOP has been addressed by dividing it into its two composing phases.
CHAPTER 1. INTRODUCTION

1.4. CONTRIBUTION OF THE THESIS

Identification of the code portions that are most suitable for migration to aspects is conducted during the aspect mining phase, in which the source code is analyzed and candidate aspects are located. Then, in the refactoring phase, the code is transformed, so that crosscutting concerns are realized by separate aspects instead of the original classes. The theoretical properties of the proposed methods are validated by a number of experiments.

During migration, human guidance is both necessary and desirable; the process requires value-judgments regarding trade-offs best made by a maintenance engineer. The process of migrating existing software to AOP is highly knowledge-intensive and any migration toolkit therefore should include the user in the change-refine loop. However, notwithstanding this inherent human involvement, there is considerable room for automation.

Among the programming languages and tools that have been developed to support AOP, AspectJ [43], an extension of Java with aspects, is one of the most popular and best supported. Even if the proposed methods and the implemented tools are based on Java and AspectJ, the approach is general and it can be extended and applied also to other languages.

1.4 Contribution of the thesis

The contribution of this thesis is the definition of a migration process for existing software systems from traditional programming paradigms (OOP) to aspect oriented programming. This contribution is articulated into many results.

- We defined a novel method to identify automatically the crosscutting concerns present in an existing Java application, *dynamic aspect mining* [62]. It is based on the analysis of the traces of use-case executions. The only assumption is that some of them exercise the crosscutting
Chapter 1. Introduction

1.4. Contribution of the Thesis

Functionalities to be separated into aspects. This corresponds to traceability from requirements (that correspond to use-cases) to implementation. In fact, whenever a requirement has a scattered and tangled implementation, it is possible to define a use-case for such a requirement, which exercise precisely the non-modularized functionality.

- The same technique has been compared with two other approaches (fan-in analysis and identifier analysis), by applying all of them to a common benchmark case [16]. They have been mutually compared and their respective strengths and weaknesses have been assessed. Moreover, by identifying where techniques overlap and where they are complementary, interesting combinations have been proposed [17]. These combinations have been applied to the same benchmark application to verify whether they perform better than the original techniques.

- We defined the notion of aspectizable interfaces [63] as those interfaces that collect transversal properties that crosscut the principal decomposition, in contrast to the more standard notion of interface which collects abstract properties of the principal decomposition, shared by all the classes implementing such an interface.

- A technique was then proposed for the aspectization of interface implementations. Identification of the interfaces that are most suitable for migration to aspects is conducted during an aspect mining phase, in which the source code is analyzed and candidate aspects are located. Then, in the refactoring phase, the code is transformed, so that interface implementations are realized by separate aspects instead of the original classes. We have implemented a toolkit to support the aspectization of interface implementations and we have applied it to the source code of some existing applications. The aim of the experi-
mental work was to assess the feasibility of the transformation and to evaluate the potential benefits [64].

- Six refactorings have been introduced to support migration [18] from OOP to AOP. They have been combined with existing OO transformations in a tool that automates them [7, 6]. The effectiveness of the approach was investigated and some case studies provide evidence suggesting that migration can be achieved with a simple set of refactoring transformations [8]. The case studies also point to the importance of enabling transformations which transform an OO program into a semantically equivalent OO program, in which the refactorings become applicable.

- All the proposed techniques have been implemented into tool prototypes which are publicly available. Such tools have been used to apply the presented methods on a large set of case study applications for a total of more than half a million lines of code. In particular, one of these applications, JHotDraw, has been subjected to all the methods. In order to measure the effect of the migration [18, 19], an empirical study has been conducted. This allowed us to assess the benefits achieved after migration, in terms of source code understandability and maintainability [64].

1.5 Structure of the Thesis

In the rest of this dissertation the migration process will be presented and applied to a number of cases.

Chapter 2 presents the current state of the art about aspect oriented programming. It reports the main approaches that have been defined to support this new programming paradigm. More attention will be paid
to the presentation of the AspectJ language, which represents the most mature and supported approach to AOP. The most relevant works about aspect mining and refactoring will be also described.

Chapter 3 contains the detailed description of the proposed methods for aspect mining. The first one is dynamic aspect mining. The steps behind its adoption will be illustrated with particular attention to the way use-cases should be selected. Formal concept analysis will be also presented, because it plays an important role in extracting concern seeds from use-case execution traces. The notion of aspectizable interfaces will be then described, together with the algorithms behind all the heuristics that are used to identify this particular kind of interfaces.

Once identified, crosscutting concerns can then be migrated into aspects using the methods described in Chapter 4. In the first part, the chapter describes which transformations are required to migrate those interfaces that have been classified as “aspectizable”. In the second part, an iterative process is presented that takes advantage of a collection of refactorings to transform more general crosscutting concerns into aspects.

Chapter 5 contains the assessment of all the proposed methods. These methods have been applied to a number of existing applications in order to show the feasibility of migration and to measure the quality of the obtained results. In particular, dynamic aspect mining has been compared with techniques proposed by other authors. The chapter contains also an empirical study that compares the obtained AOP code with the original OOP code in terms of both external and internal qualities.

Finally, Chapter 6 concludes the thesis by drawing conclusions from the presented results. Future works are also discussed.

Appendix A reports the formal definitions of the six proposed refactorings presented in Chapter 4.
Chapter 2

State of the Art

In this chapter a survey on the state of the art in aspect oriented programming is given. The chapter is divided into four sections. In the first section AOP is introduced focusing on the key idea of inversion of dependency. The second section, devoted to AOP development, starts with an overview on the different languages proposed to support AOP. After that, research areas are described in support to the requirement and design phase. An overview of the static analysis of aspects is also given. The third and fourth sections deal with a description of existing approaches supporting migration from OOP to AOP. These works are actually the most related to the subject of the thesis. They include aspect mining and refactoring.

2.1 AOP

Aspect Oriented Software Development (AOSD) is a programming paradigm in which crosscutting concerns that pervade a system are isolated and extracted into separate modules called aspects. Unlike traditional modules, which must be referenced explicitly by the program (e.g., via procedure call), aspects add functionalities to the base code by intercepting the execution flow, without any need for the base code to mention the aspect code explicitly. In other words, the base code remains oblivious to the function-
ality added by an aspect, and it is the aspect that specifies (quantifies) the places in the base program affected by the new functionality [25]. Examples of crosscutting functionalities that benefit from the obliviousness and quantification mechanisms offered by AOSD include persistence, logging, caching, transaction management and contract enforcement.

The motivation which underpins the AOSD paradigm is that these crosscutting functionalities are scattered throughout the program and tangled with the base functionalities, with a corresponding detrimental effect upon many software engineering activities, such as program comprehension, maintenance, and evolution. In the AOSD paradigm, the base code that implements the core functionality of the system is disentangled from and oblivious to the crosscutting aspect code, thanks to the quantification constructs available in aspects. It is expected that this form of disentanglement of base code from aspect code will allow the two to be understood, developed, tested and maintained independently from one other, with the base code free of references to any other crosscutting concern.

In order to extend the benefits of AOSD to existing systems, a significant reverse and re-engineering effort will be required. Moreover, the execution of reverse and re-engineering studies is expected to contribute to the clarification of the applicability of AOSD to real systems, as well as to the refinement of the AOP constructs and concepts themselves. This reverse and re-engineering effort will need to identify and extract from existing programs the code which denotes the crosscutting functionality and to refactor these existing programs within an aspect-oriented re-formulation of the original program.

The term “migration” indicates the whole process used to move an object-oriented application to be aspect-oriented, so as to exploit the novel constructs and code structuring opportunities offered by the aspects. “Refactoring” indicates a specific activity within migration, which consists of
small, controlled and largely automated program transformations, resulting in a step forward in the migration process. As with other migrations and conversions [15, 44, 67, 69], the refactorings involved in migrating from OOP to AOP must be automated to avoid unsustainable cost.

The problem of identifying candidates for aspects in code has been well studied in the literature, as discussed in a subsequent section, while, the problem of extracting aspects from an OO program in order to refactor it into an equivalent AOP program is much less well–studied in the literature.

2.2 Approaches

2.2.1 Aspect Oriented Development

Most of the works in the aspect oriented literature fall into the area of (initial) development. They range from the programming languages to use, to requirement specification, design and static analysis. Migration was instead somehow neglected.

Composition Filters

The composition Filters model was originally introduced to integrate database features into an object oriented language without losing important language features, such as encapsulation, polymorphism and delegation. This work was later extended [5] to cope with AOP.

The composition filters approach achieves the separation of the cross-cutting concerns from the principal module decomposition by providing new language features. Those functionalities that crosscut the base modularization and that do not match any object model element are encoded as composition filters.

When a message is received by an object, it is handled by a dispatcher that matches the message against the filters defined upon the object. Each
filter accept function is evaluated to determine whether the filter applies or not. If the message matches the accept function, it is dispatched to the new target that is specified in the filter function body. The new target can be a method defined in the object, or another method coming from interface objects.

Composition filters provide a separated modularization for crosscutting concerns because the implementation of a filter is completely independent from the objects on which the filter is super-imposed. Moreover the implementations of different concerns are separated and orthogonal, because a filter specification does not refer to other filters.

Generative Programming

Smith [58] presented an approach to the problem of the appropriate modularization of crosscutting concerns, based on generative programming. The central idea is to express crosscutting concerns as logical invariants that must be verified by the system implementation at run time. Code fragments are automatically generated to modify the program behavior at all the points in the code where the invariants are possibly violated. The aim of the generated code is to restore the invariant validity.

The generative programming method applies to already existing systems in which a new feature has to be added, when the feature can be expressed as a logical function. Developers have to translate the textual concern description into an invariant that gives a declarative semantical definition of the new feature.

The behavior of a program is modeled as a sequence of states, and the switch from a state to another one is triggered by an action. The invariant is a logical function that can be evaluated in every state. The invariant uses the program variables and composes them using logical connectors.

All the points in the program that possibly violate the invariant are
called *disruptive points*. In order to locate all these points in the system, a static code analyzer looks for the changes of the variables that are used by the invariant functions. For each potentially disruptive action, the code analyzer generates action-specific maintenance code, in order to keep the invariants verified.

**Hyper-slices**

This approach recognizes that the major cause of the maintainability, reusability and traceability problems is the *tyranny of the dominant decomposition*. Existing modularization mechanisms typically support only a single *dominant* decomposition. Thus there is no support for those changes that require to reason about the application in dimensions different from the principal one.

The approach presented in [60] aims at breaking this tyranny by proposing a model that allows simultaneous, multi-dimensional decomposition and composition of the software modules.

The new unit of decomposition is the *Hyper-slice*. A Hyper-slice is a set of conventional modules, written in any programming language (e.g in Java). The modules within a Hyper-slice contain all, and only, those parts that pertain to, or address, a given concern. For example the persistence hyper-slice contains all those pieces of code coming from different classes, that deal with writing and retrieving the class data to/from a persistent storage.

Joining all the slices into a single whole is not a fully automatic process, because of the presence of overlapping and conflicting parts. In such cases developers have to solve mismatches in order to produce the final result.
2.2. APPROACHES

Aspect Oriented Languages

The most popular way to apply AOP in software development is to extend existing programming languages with new constructs, the most important of which are pointcuts, advices and introductions, that make it possible to express the crosscutting functionalities. There are aspect versions of many widely used languages as, for instance, Java, C/C++, Ada, Smalltalk, Perl, Python, etc. All of them share, more or less, the same architecture, that was introduced by the aspect extension for Java, AspectJ [42].

An aspect can intercept those points in the execution flow where the concern is required. These points are called *join points*. When the application reaches a join point, the aspect takes control and executes the concern code. Then, control is returned to the application.

The executable application is produced by a so called *weaver*, which merges the aspect code and the base code together. It resolves the pointcuts, adding calls to the corresponding aspect code. With AspectJ, the result of the weaving process is a Java compatible application in which aspects and classes coexist in the produced byte-code. A more detailed description of AspectJ will follow.

Transformation tools

Aspect Oriented programming can be interpreted as a special case of code-to-code transformation, where the transformation itself is specified by the aspect code, the transformation engine is the weaver and the subject of the transformation is the code in the principal decomposition.

Legacy systems, developed using very old languages, could benefit from the usage of the more versatile decomposition offered by AOP. Unfortunately, the effort required to develop an aspect weaver for such languages can be too hard, especially when there exist several different dialects of
the same language. Instead of developing a different aspect weaver for any different language, program transformation engines provide an alternative solution.

**Frameworks**

Aspect oriented frameworks provide an implementation of AOP without requiring the adoption of new language constructs. For example, JAC [50] (Java Aspect Components) is a framework for aspect oriented programming in Java. An aspect program in JAC is a set of aspect objects that can be dynamically deployed/undeployed on top of running application objects.

The *weaver* is a regular java program that uses introspection features to allocate aspect objects to the appropriate base objects. Being executed at run-time, it can add/remove aspect objects dynamically.

The main advantage of aspect frameworks is that no syntactic extensions are required. Base, aspect and weaver classes define their roles by extending particular classes in the framework. The framework itself takes care of handling them, performing the composition in the appropriate way and when required.

**Requirements**

To support system evolution in response to changed requirements, it is important to think about and identify the crosscutting concerns in the early stage of the system development. In [53] the notion of *early aspects* is introduced for those aspects that are identified at the requirement level. In this work, authors cope with the definition of a model that is able to separate cross-cutting functional and non functional properties at the requirement engineering level. The first step in the definition of early aspects is to discover the requirements, split them into detailed, small parts, the *viewpoints*, and relate them to concerns.
A concern that involves several requirements, or that influences or contains more than one viewpoint is considered a candidate aspect. Each candidate aspect must be specified in detail in order to detect interactions and conflicts.

**Design**

In order to identify and model a broad range of aspects early in the software development life cycle, and assess where they crosscut the system, developers need support for analyzing the relationships among the requirements. They also need support to translate the results of this analysis into design models which can then be translated into code.

The Theme [1] approach supports aspect oriented development by providing a UML-like design language. In this approach a theme is an element of the design that contains a collection of structures and behaviors that represent a single feature. The whole system comes out of the combination of several themes. There are two types of themes, base themes and crosscutting themes. *Base themes* may share structure and behavior with other themes, modeling them from their own perspective. *Crosscutting themes* have behavior that overlaps with the functionality of the base themes. This second type of themes will be mapped to aspects.

In [59], using standard UML extension mechanisms UML is extended as a design notation for AspectJ. This notation provides a representation for all the component of an aspect (join points, pointcuts, advices, introductions). Aspects can, thus, be fully specified in a UML design model, bringing the advantages of the separation of the concerns also at the design level.
Static Analysis

After an application has been developed, using aspect oriented programming, and released, the life cycle goes on with maintenance. Since the paradigm is still quite new there are only a few techniques that can be applied to support software comprehension and evolution.

Inter-procedural static slicing has been adapted to aspect oriented software in [70]. The computation of a program slice starts with the definition of a slicing criterion. This criterion contains a variable from the program and a line of code containing this variable. A program slice is a subset of the program instructions that is equivalent to the initial program, with respect to the slicing criterion. This means that if the original program and the slice are executed on the same input value, they will produce the same output value for the variable in the criterion at the line of code specified in the criterion (assuming termination).

Program slicing has many applications in software engineering activities including program comprehension, debugging, testing, maintenance and model checking. In fact, a slice is easier to understand, because it contains fewer instructions than the original code, including only those that are required for the computation of the variable of interest at the given line of code.

A pointcut designator defines the portions of code to be intercepted by an aspect, by specifying which conditions on the call stack must be verified in order for the aspect to apply. Since the call stack contains run-time informations that are not available at weave time, pointcut designators can not be fully resolved. Therefore, instructions are inserted at the join points to evaluate such conditions at run time, producing a computation overhead.

Static analysis of aspect oriented code is investigated in [55], to resolve
pointcut designators at weave time, thus improving aspect code performance. The objective is to determine a conservative approximation of all the possible call stacks for each procedure call.

2.2.2 An AspectJ primer

Amongst the programming languages and tools that have been developed to support AOSD, AspectJ [43], an extension of Java with aspects, is one of the most popular and best supported. AspectJ offers two main programming constructs to modularize crosscutting concerns: pointcuts and introductions. *Pointcuts* intercept the normal execution flow at specified *join points*. Aspect code (called *advice*) can be executed either before, after, or instead of (around) the intercepted join points. *Introductions* modify properties of the classes they affect by adding methods or fields and by making them implement interfaces or specialize super-classes previously unrelated to them. Unlike pointcuts/advices, which alter the dynamic behavior, introductions operate statically on the class members and structure (this may, in turn, have effects on the dynamic behavior in the presence of dynamic method binding).

Fig. 2.1 shows an example aspect that adds persistence to an existing class, *Person*. Similar to classes, aspects have fields and methods. The aspect *PersistentPerson* declares two fields, *outfile* and *out*, respectively of type *FileOutputStream* and *ObjectOutputStream*. Next, the aspect *PersistentPerson* declares that the interface *Serializable* is implemented by class *Person*. This is achieved by means of the AspectJ construct *declare parents*. Moreover, the aspect adds related methods *readObject* and *writeObject* to the current implementation of class *Person*. The syntax for the necessary introductions consists of the base class name, dot-separated by the added method (field) name (e.g., *Person.readObject*).
In order to serialize all objects of class `Person` as soon as they are created, the appropriate join points are all calls to any constructor of class `Person`. The pointcut `personCreation` intercepts such join points. The AspectJ keyword `call`, followed by a method (constructor) signature, is used to specify a primitive pointcut which intercepts all calls to the methods (constructors) matching the given signature. Parameter passing between base code and aspect code is achieved through context exposure. The AspectJ keyword `target` is used to expose the object on which the invocation is performed (in this case, the newly created `Person` object). The exposed context is then available within the advices associated with the pointcut.

The advice to be executed after the pointcut `personCreation` is declared by means of the AspectJ keyword `after`. Its formal parameters match those exposed in the corresponding pointcut. The code in the after-advice defined for the pointcut `personCreation` stores information about
the `Person` object being created by calling the method `writeObject` on the aspect field `out` and passing the `Person` object to it. In turn, the method `writeObject` is known to delegate the write operation to its parameter object, of type `Serializable`, by invoking its `writeObject` method.

Among the other primitive pointcuts provided by the AspectJ language, `execution` and `handler` are relevant for the present work. `Execution(<signature>)` intercepts the execution of any method matching `<signature>`, while `handler(<exception>)` matches any exception handler (catch block) matching `<exception>`. The primitive pointcut `withincode` is satisfied by join points inside the code specified after the keyword (e.g., `withincode(Person.new(String))`). The primitive pointcut `if` is used to express a condition to be checked dynamically. It intercepts all join points making the condition true. Wildcards can be also included in pointcut specifications (e.g., `call(* Person.*(..))` indicates a call to any method of class `Person` with any signature).

Advices executed `after`, `before`, or `instead` of a given join point are specified through the keywords `after`, `before`, and `around`, respectively. After-advises can be further distinguished, depending on the kind of “return”: normal termination is specified as `after() returning` and exceptional termination as `after() throwing`. Inside an around-advice, the original computation can be executed using `proceed()`. Around-advises are expected to return a value of the same type as the intercepted join point.

In addition to `target`, the primitive constructs provided by AspectJ for context exposure include: `args` (to expose the arguments of a method in a `call` or `execution` pointcut) and `this` (to expose the current object). By default, a single aspect instance is created for each aspect declaration. However, it is possible to change the default by means of a per-clause, option such as `pertarget`, `perthis`, `percflow`, whenever fresh aspect fields are required, respectively, each time a new target, `this` object or control
flow matching a given pointcut are encountered.

A useful mechanism provided by AspectJ to support aspectization of exception handling is the \texttt{declare soft} construct, which softens an exception so that it needs not be caught inside the base code and it can be managed completely inside an aspect. An example of its usage will be provided in Chapter 4. More details on the semantics of AspectJ can be found in the Programming Guide accompanying its distribution (http://aspectj.org) as well as any of the many books published on the subject (e.g. [43]).

### 2.3 Aspect Mining

Legacy systems can have potentially relevant benefits from the separation of concerns realized by means of aspects. The first step in such a process is the identification of the crosscutting functionalities to be implemented as aspects. Such a task is called \textit{aspect mining}.

Some of the various aspect mining approaches rely upon the user definition of likely aspects, usually at the lexical level, through regular expressions and support the user in the code browsing and navigation activities conducted to locate them. In a typical adoption scenario the existing system knowledge suggests a staring point that represents a portion of a candidate crosscutting concern (\textit{aspect seed}). Code browsing tools aid the user in navigating through the source code to further expand the initial concern. The Aspect Mining Tool AMT [36] supports aspect identification by matching textual patterns against the names used in the code and by looking for repeated uses of the same types. The Aspect Browser [30] tool also uses textual patterns to match the aspects. Their location is improved by adopting a map-based display where aspects are shown in colors. The code browsing tool JQuery is presented in [41]. JQuery provides hierarchical navigation and query facilities, which are useful while executing aspect
extraction tasks. Concern graphs [54] can be employed to effectively represent crosscutting concerns. These graphs show the classes, methods and fields involved in a concern and their mutual relationships. They are built incrementally during source code exploration.

Other approaches try to improve the identification by adding more automation to analyze the source code and compute one or more aspect candidates.

In [14] the problem of identifying some known concerns in industrial code has been investigated using clone detection. Several different clone detection algorithms have been applied. Existing crosscutting concerns have been identified by manual inspection and then compared with the results of each clone detection technique. Such a comparison represents an assessment of the suitability of clone detection as an aspect mining technique.

Dynamic information [10] was used for aspect mining. Program execution traces are analyzed to discover several different kinds of execution relations among methods. Recurring patterns in method invocations are then identified. When such patterns are crosscutting with respect to the program main decomposition, the patterns are candidates to be turned into an aspect advice.

Natural Language Processing (NLP) was exploited [57] to identify crosscutting concerns, under the assumption that these are implemented following proper naming conventions. NLP is used to analyze all the natural language clues that developers insert in comments and source code. All the extracted information is represented in the Action-Oriented Identifier Graph (AOIG). Such a graph models the relation between verbs and nouns using the so called verb-DO pairs, verb and noun pairs where the verb represents an action (operation) and the noun is the object of the action. This graph has been used to build a tool supporting not only aspect mining, but
also feature location and working set recovery.

Clustering [56, 39] has been used to group together methods that belong to a crosscutting concern. Clustering starts by assigning each method to a distinct cluster. Then, clusters with the smallest distance are recursively merged together into a new cluster. In [56] the distance function is based on the presence of common substrings in method names, whereas in [39] distance is evaluated using the Static Direct Invocation Relationship (SDIR) among methods: a smaller distance is associated with methods that are frequently called together, whereas a higher distance identifies those that are called together not so often.

A simple heuristics, unique method [32], has been used to implement an aspect mining method. When AOP languages were not available, developers used to implement typical crosscutting concerns (such as logging or notification) in a single method that was called from many scattered places in the code. In this way they avoided tangling between the concern and the base code, but scattered calls still remain as well as a dependency from the base (non-oblivious) code to the concern code. A unique method is defined as “a method without a return value which implements a message implemented by no other method” [32]. Static analysis is used to detect all the methods that satisfy this definition. They are then sorted according to the number of times they are called and eventually filtered. Even if this approach seems quite simple, it has been able to find typical crosscutting concerns in a Smalltalk case study application.

**Fan-in analysis**

The *fan-in* metric, as defined by Henderson-Sellers, counts the number of locations from which control is passed into a module [40]. In the context of object-orientation, the module-type to which this metric is applied is the method. The *fan-in* of a method $M$ has been defined in [48] as the number
of distinct method bodies that can invoke $M$.

Typical examples of crosscutting concerns include logging, tracing, pre- and post-condition checks, and exception handling. It is exactly this type of crosscutting that fan-in analysis tries to capture. The hypothesis is that the amount of calls to a method implementing this crosscutting functionality (fan-in) is a good measure for the importance and scattering of the discovered concern.

The fan-in aspect mining process consists of three steps:

1. Automatic computation of the fan-in metric for all methods in the investigated system.

2. Filtering of the results from the previous step by eliminating all methods with fan-in values below a chosen threshold, eliminating the accessor methods (whose name matches a `get*/set*` pattern) and eliminating utility methods (like `toString()` and collection manipulation methods).

3. (Partially automated) analysis of the methods in the resulting, filtered set by exploring the callers, call sites, naming convention used, the implementation and the comments in the source code.

The result of the fan-in analysis is a set of candidate seeds, represented by methods with high fan-in.

**Identifier analysis**

In the absence of designated language constructs for aspects, naming conventions are the primary means for programmers to associate related but distant program entities. This is especially the case for object-oriented programming, where polymorphism allows methods belonging to different classes to have the same signature. Moreover, it is good practice to use
intention-revealing names [4]. Design and programming patterns provide a common vocabulary shared by programmers.

.Identifier analysis [66] relies on this assumption and identifies candidate seeds by grouping program entities with similar names. More specifically, it applies Formal Concept Analysis (FCA, see Subsection 3.1.1) with all classes and methods as FCA-objects (except those that generate too much noise in the results, like test classes and accessor methods), and the identifiers associated with those classes and methods as FCA-attributes.

The identifiers associated with a method or class are computed by splitting up its name based on capitals (camel-casing). For example, a method named createUndoActivity yields three identifiers create, undo and activity. In addition, the Porter stemming algorithm [51] was applied to make sure that identifiers with the same root form (like undo and undoable) are mapped to one single representative identifier or “stem”. It is these stems that are used as FCA-attributes for the concept analysis.

The FCA algorithm then groups entities with the same identifiers. When such a group contains a certain minimum number of objects (classes and methods) and the entities contained in it cut across multiple class hierarchies, the group is considered a candidate aspect seed. The only remaining (though most difficult) task is that of deciding manually whether a candidate seed is a real seed or a false positive.

2.4 Refactoring

Software refactoring consists of the modification of the internal program structure without altering the external behavior (semantic preservation). It aims at improving internal quality factors, such as modularity, in order to make the code easier to understand and evolve in the future. In the migration of existing OO code to aspects, the problem that has received
most attention is the detection of aspect candidates (aspect mining). The problem of refactoring has been considered only more recently.

Traditional object oriented refactoring can not be directly applied to aspect oriented systems because the behavior is not guaranteed to remain preserved. In fact, a change in a base object could make some pointcuts not longer valid, thus modifying the concern semantics. The relationship between object-oriented refactoring and AOP is discussed in [33]. Moreover, popular OO transformations taken from Fowler [26] has been re-defined in order to make them aspect-aware. Novel refactorings have been also defined to migrate crosscutting concerns into aspects and to restructure aspect code.

Catalogs of refactorings [49, 21] to migrate object-oriented code into aspect-oriented have been defined together with refactorings that apply to aspect code. The feasibility of the approach has been shown on small case studies, such as the observer design pattern [49]. Behavior preservation conditions for aspect refactorings are also formally specified [21].

Even small changes in the base code could make a pointcut fail in capturing all the intended join points. This problem is known as the *fragile pointcut* problem. Pointcuts are particularly fragile when they are written in an enumerative style, whereas an intensional pointcut is expected to be more robust. Inductive logic programming[31] has been used to transform an extensional definition of a pointcut (that merely enumerates all the join points), into an intensional one.

Automated support for transforming object-oriented code into aspects has been investigated [24, 38]. An approach derived from program slicing [24] has been implemented to improve a popular object oriented refactoring, *method extraction*. A generalization of this refactoring has been defined to untangle the crosscutting concerns from the base code and to move them into an aspect. The notion of the role played by the classes
involved in the implementation of design patterns [38] has been used as the basis of a refactoring tool. A library contains some predefined crosscutting concerns together with those refactorings that should be applied to change each particular concern into an aspect. Both the crosscutting concern and the refactoring are specified in terms of abstract roles. A human intensive effort is required to manually map abstract roles to concrete program elements. Once the mapping is complete the actual refactoring process proceeds with a high degree of automation.

In [47] the Undo concern present in JHotDraw has been manually refactored from Java to AspectJ. The goal was to (manually) understand the degree of tangling between the Undo concern and the base code. Preliminary OO transformations have been applied to reduce tangling, so as to produce code which is easier to migrate.

There is a number of code duplication patterns that object oriented modularization is not able to factor out. For instance, using the java language, exception handling requires a fixed (try/catch blocks) structure that must be repeated each time the same exception has to be managed. Other examples are concurrency control, worker object creation, argument trickle, lazy initialization and contract enforcement. An approach to refactor these patterns using AOP has been proposed [45]. This approach is intended to solve specific problems that affect a small portion of the system, typically starting by refactoring a single class.
Chapter 3

Aspect Mining

In this chapter, two aspect mining approaches are presented. The first approach relies on a dynamic analysis of the application which runs on selected execution scenarios. The second method is based on the notion of roles attached to particular kinds of interfaces, called aspectizable interfaces.

3.1 Dynamic aspect mining

In this section, the dynamic aspect mining approach is described after a background introduction of the dynamic feature location method. The tool implementation concludes the section.

3.1.1 Dynamic feature location

The method for feature location based on concept analysis is summarized here, in order to cover the background required to understand the proposed aspect mining technique. The interested reader can find more details in [23].

Formal concept analysis (FCA) [28] is a branch of lattice theory that
can be used to identify meaningful groupings of objects\textsuperscript{1} that have common attributes.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming lang.</td>
<td>object-oriented</td>
</tr>
<tr>
<td>Java</td>
<td>√</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>√</td>
</tr>
<tr>
<td>C++</td>
<td>√</td>
</tr>
<tr>
<td>Scheme</td>
<td>-</td>
</tr>
<tr>
<td>Prolog</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.1: Programming languages and their supported programming paradigms.

FCA takes as input a so-called context, which consists of a (potentially large, but finite) set of objects $O$, a set of attributes $A$ on those objects, and a Boolean incidence relation $R$ between $O$ and $A$. An example of such a context is given in Table 3.1, which relates different programming languages and attributes. A mark √ in a table cell means that the object (programming language) in the corresponding row has the attribute of the corresponding column.

Starting from such a context, FCA determines maximal groups of objects and attributes, called concepts, such that each object of the group shares the attributes, every attribute of the group holds for all of its objects, no other object outside the group has those same attributes, nor does any attribute outside the group hold for all objects in the group. Intuitively, a concept corresponds to a maximal ‘rectangle’ containing only √ marks in the table, modulo any permutation of the table’s rows and columns.

Formally, the starting context is a triple $(O, A, R)$, where $R \subseteq O \times A$ is a binary relation between the set of all objects $O$ and the set of all considered object attributes $A$. A concept $c$ is defined as a pair of sets $(X, Y)$ such

\textsuperscript{1}Not to be confused with objects in Object-Oriented programming.
that:

\[ X = \{ o \in O \mid \forall a \in Y : (o, a) \in R \} \]  

\[ Y = \{ a \in A \mid \forall o \in X : (o, a) \in R \} \]

where \( X \) is said to be the *extent* of the concept \( (\text{Ext}[c]) \) and \( Y \) is said to be its *intent* \( (\text{Int}[c]) \). It should be noticed that the definition above is not “constructive”, being mutually recursive between \( X \) and \( Y \). However, given a pair \((X, Y)\), it allows deciding whether it is a concept or not. FCA algorithms provide constructive methods to determine all pairs \((X, Y)\) satisfying the constraints (1) and (2).

![Figure 3.1: The concept lattice for Table 3.1.](image)

The containment relationship between concept extents (or, equivalently, intents) defines a partial order over the set of all concepts, which can be shown to be a lattice [29]. Figure 3.1 shows the concept lattice corresponding to Table 3.1. A concept is represented by a node, concepts are connected by an edge when the lower concept is a sub concept of the upper one. In a node, the first row reports the concept attributes (intent) while the second rows contains the concept objects (extent). The lattice’s bot-
tom concept contains those objects that have all attributes. Since there is no such programming language in our example, that concept contains no objects (its extent is empty). Similarly, the top concept contains those attributes that hold for all objects. Again, there is no such attribute (the concept’s intent is empty). Other concepts represent related groups of programming languages, such as the concept $c_3 (\{\text{Java, C++}\}, \{\text{static typing, OO}\})$, which groups all statically-typed object-oriented languages, a sub-concept of all OO languages. Intuitively, the sub-concept relationship can thus be interpreted as a specialization of more general notions. Underlined objects (resp. attributes) are those that are most concept-specific, being attached to the largest lower bound (resp. least upper bound) concept. When using the so-called sparse labeling of the concept lattice, only these underlined labels are retained, without loss of information.

Complete information about each node $n$ in the concept lattice $L$ is given by the pair $(\text{Ext}[n], \text{Int}[n])$. However, it is possible to represent the same information in a more compact and readable form by marking a node $n$ with an object $o \in \text{Ext}[n]$ or an attribute $a \in \text{Int}[n]$ only if it is associated with the most special (respectively, general) concept $c$ having $o$ (resp., $a$) in the extent (resp., intent). The (unique) node of $L$ marked with a given object $o$ is thus:

$$\gamma(o) = \inf\{n \in L| o \in \text{Ext}[n]\} \quad (3.3)$$

where $\inf$ gives the infimum (largest lower bound) of a set of concepts. Similarly, the unique lattice node marked with a given attribute $a$ is:

$$\mu(a) = \sup\{n \in L| a \in \text{Int}[n]\} \quad (3.4)$$

where $\sup$ gives the supremum (least upper bound) of a set of concepts. The objects in the extent of a lattice node $n$ are then obtained as the set of
objects at or below \( n \), while the attributes in its intent are those marking \( n \) or any node above \( n \) (see Figure 3.2).

Figure 3.2: The concept lattice for Table 3.1, with sparse labeling.

The labeling introduced by the functions \( \mu \) and \( \gamma \) give the concept most specific of a given attribute/object. Thus, given a concept \( c \), the objects and attributes that label it indicate the most specific properties (objects and attributes) that characterize it. Sometimes it is convenient to get the labels of a given concept through the following functions:

\[
\alpha(c) = \{ a \in A \mid \mu(a) = c \} \quad (3.5)
\]

\[
\beta(c) = \{ o \in O \mid \gamma(o) = c \} \quad (3.6)
\]

\( \alpha(c) \) gives the set of attributes labeling a concept \( c \), while \( \beta(c) \) gives the concept’s objects in the \( \gamma \) labeling.

The goal of feature location [23] is to identify the computational units (e.g., procedures, class methods) that specifically implement a feature (e.g., requirement) of interest. Execution traces obtained by running the program under given scenarios provide the input data (dynamic analysis).
3.1 DYNAMIC ASPECT MINING

Concept analysis is applied to a context where attributes are computational units, objects are scenarios and the relationship $R$ contains the pair $(o, a)$ if the computational unit $a$ is executed when scenario $o$ is performed. Information about which computational unit is executed for each scenario is obtained by gathering execution traces of the program.

A concept in the resulting concept lattice groups all computational units executed by all scenarios in the extent. Moreover, a computational unit labels a given concept if it is the most specific computational unit for the scenarios in the concept extent. Assuming that each scenario is associated with exactly one feature (the general case of a many-to-many relationship between scenarios and features is dealt with in [23]), the concept specific of a given feature is the one (if any exists at all) which has only the associated scenario in its extent (e.g., concept $c_6$ for the scenario Prolog in Figure 3.2). Correspondingly, the computational units specific for a given feature are those which label such a concept (Logic for $c_6$).

When a feature-specific concept does exist, the attributes that label it are the most specific computational units involved in the execution of the scenario in its extent. In other words, they represent the code portions that are most specifically devoted to implementing the functionality exercised by the scenario in the concept extent.

3.1.2 Dynamic aspect mining technique

The first step in the migration of existing applications to AOP consists of identifying the crosscutting concerns that are amenable for an aspect-oriented implementation. Such a process can be driven by the use cases of the application. In fact, each use case specifies a functionality of the system. When such a functionality is implemented by code fragments spread across several modularization units, it is possible to turn it into an aspect. In the restructured code, each distinct functionality will be located in ex-
CHAPTER 3. ASPECT MINING

3.1. DYNAMIC ASPECT MINING

Exactly one modularization unit. However, benefits in program understanding and evolution can be actually achieved only if the new modularization units that factor out the crosscutting concerns are relatively independent (decoupled) from the other units. In other words, restructuring should aim both at separating the crosscutting concerns (increased intra-module cohesion) and at untangling them from the original code (reduced inter-module coupling).

We propose to use feature location for aspect mining according to the following procedure. Execution traces are obtained by running an instrumented version of the program under analysis for a set of scenarios (use cases). The relationship between execution traces and executed computational units is subjected to formal concept analysis. The execution traces associated with the use cases are the objects of the concept analysis context, while the executed class methods are the attributes. In the resulting concept lattice (with sparse labeling), the use-case specific concepts are those labeled by at least one trace for some use-case (i.e. \( \alpha \) contains at least one object), while the concepts with zero or more attributes as labels (those with an empty \( \alpha \)) are regarded as generic concepts. Thus, use-case specific concepts are a subset of the generic ones.

Both use-case specific concepts and generic concepts carry information potentially useful for aspect mining, since they group specific methods that are always executed under the same scenarios. When the methods that label one such concept (using the sparse labeling) crosscut the principal decomposition, a candidate aspect is determined.

Formally, let \( C \) be the set of all the concepts and let \( C_s \) be the set of use-case specific concepts (\(|\alpha(c)| > 0\)). A concept \( c \) is considered a candidate seed iff:

**Scattering:** \( \exists a, a' \in \beta(c) \mid \text{pref}(a) \neq \text{pref}(a') \)
Tangling: \( \exists a \in \beta(c), \exists c' \in \Omega, \exists a' \in \beta(c') \mid c \neq c' \wedge \text{pref}(a) = \text{pref}(a') \)

where \( \Omega = C_s \) for the use-case specific seeds, while \( \Omega = C \) for the generic seeds. The first condition (scattering) requires that more than one class contributes to the functionality associated with the given concept (\( \text{pref}(a) \) is the fully scoped name of the class containing the method \( a \)). The second condition (tangling) requires that the same class addresses more than one concern.

In summary, a concept is a candidate seed if:

- **scattering**: more than one class contributes to the functionality associated with the given concept;

- **tangling**: the class itself addresses more than one concern.

The first condition alone is typically not sufficient to identify crosscutting concerns, since it is possible that a given functionality is allocated to several modularized units without being tangled with other functionalities. In fact, it might be decomposed into sub-functionalities, each assigned to a distinct module. It is only when the modules specifically involved in a functionality contribute to other functionalities as well (i.e. the second condition) that crosscutting is detected, hinting for a candidate seed.

Aspectization of the crosscutting concerns is one of the possible actions to remedy the scattering problem detected by means of feature location. Sometimes, more standard refactoring [26] actions may be sufficient. For example, if a class has too many responsibilities, being involved in several use-cases with specific methods, it might be possible to extract some of its methods/fields, or to build a sub-class out of a part of it (see “move method, move field and extract sub-class” in [26]), thus achieving a better distribution of the responsibilities. When the crosscutting functionality cannot be modularized by means of standard refactoring techniques, AOP is an option.
CHAPTER 3. ASPECT MINING

3.1. DYNAMIC ASPECT MINING

3.1.3 Clarifying example

Let us consider a binary search tree application consisting of the two classes depicted in Figure 3.3. It supports the insertion of nodes and the search of information inside the tree. During the execution of these operations, it is possible to log the related data into a file (logging enabled) or to perform them without logging (logging disabled). These two possibilities are associated with two use-cases defined for this program. The methods executed in the two use-cases are traced. They are reported in Table 3.2.

By applying concept analysis to the relationship in Figure 3.4 (top), the concept lattice at the bottom is obtained. The bottom node in the lattice represents the concept specific of the use case with logging enabled. Its two attributes are associated with methods from different classes (scattering), which appear also in other use-case specific concepts (tangling). Thus, both conditions for the identification of an aspect candidate are met and the two methods `BinaryTree.log()` and `BinaryTreeNode.log()` are reported as the seeds associated with the identified aspect candidate.
### Tree construction with logging enabled

- $m_1$: `BinaryTree.BinaryTree()`
- $m_2$: `BinaryTreeNode.BinaryTreeNode(Comparable)`
- $m_3$: `BinaryTree.insert(BinaryTreeNode)`
- $m_4$: `BinaryTree.log()`
- $m_5$: `BinaryTreeNode.insert(BinaryTreeNode)`
- $m_6$: `BinaryTreeNode.log()`
- $m_7$: `BinaryTree.search(Comparable)`
- $m_8$: `BinaryTreeNode.search(Comparable)`
- $m_9$: `BinaryTreeNode.log()`

### Tree construction with logging disabled

- $m_1$: `BinaryTree.BinaryTree()`
- $m_2$: `BinaryTreeNode.BinaryTreeNode(Comparable)`
- $m_3$: `BinaryTree.insert(BinaryTreeNode)`
- $m_5$: `BinaryTreeNode.insert(BinaryTreeNode)`
- $m_7$: `BinaryTree.search(Comparable)`
- $m_8$: `BinaryTreeNode.search(Comparable)`

Table 3.2: Execution traces for a use case with logging enabled (resp. disabled).
3.1.4 Tool support

A tool, Dynamo\textsuperscript{2}, has been implemented to trace method executions of Java programs. ToscanaJ\textsuperscript{3} has been used for concept lattice construction and visualization.

Dynamo provides facilities to instrument the application under analysis. At the beginning of each method a probe is inserted. When such a method is called, the probe is executed and a new line is appended into the trace file. The tracing tool is based on the meta-object protocol provided by OpenJava\textsuperscript{4}. It includes a facility to switch tracing on and off during execution, thus, it is possible to trace only the portions of execution in which the functionality of interest is actually exercised, skipping, for example, any set-up and tear-down phase. With such a facility in place,

\begin{center}
\begin{tabular}{|l|c|c|c|c|c|c|c|}
\hline
 & $m_1$ & $m_2$ & $m_3$ & $m_4$ & $m_5$ & $m_6$ & $m_8$ \\
\hline
Logging enabled & $\times$ & $\times$ & $\times$ & $\times$ & $\times$ & $\times$ & $\times$ \\
Logging disabled & $\times$ & $\times$ & $\times$ & $\times$ & $\times$ & $\times$ & $\times$ \\
\hline
\end{tabular}
\end{center}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{concept_lattice.png}
\caption{Concept lattice for the binary search tree application.}
\end{figure}

\textsuperscript{2}http://star.itc.it/dynamo/
\textsuperscript{3}http://toscanaj.sourceforge.net/
\textsuperscript{4}http://openjava.sourceforge.net/
the assumption that a use-case corresponds to exactly one feature to be located becomes a reasonable one. As soon as a trace is available for each execution scenario, Dynamo generates the formal concept that ToscanaJ processes to construct the concept lattice. Such a lattice is, eventually, filtered by Dynamo and, when both tangling and scattering occur, an aspect candidate is reported.

### 3.1.5 Contribution to the state of the art

Most of the techniques in the aspect mining literature (Section 2.3) take into account static information to reveal aspect candidates, looking for replicated code fragments (clones) [14], or analyzing the words used in the code text through Natural Language Processing [57]. Other techniques cluster together portions of the application based on similarity among method names [56] or based on recurring calling relationships [39] among methods. Ad-hoc properties were also used to reveal aspect candidates, such as methods which satisfy the *unique method* heuristics [32] and methods which are called many times (high fan-in) [48].

Similarly to our approach, in [10] dynamic information is used for aspect mining, but the tool developed by the authors of [10] looks for recurring sequences of method invocations that may be turned into aspect advices. No attempt is made to draw a connection between execution scenarios and crosscutting functionalities.

Naming convention is used to identify crosscutting concerns in [66]. Similarly to us, this approach apply formal concept analysis to group together entities that belong to the same crosscutting concern, program entities whose names use common identifiers are reported as candidate aspects.

The technique presented in this section is the first attempt to let the requirements (translated into executable use-cases) guide aspect identification by means of formal concept analysis, applied to the execution traces.
The main advantages over the existing approaches is that we do not rely on naming/coding conventions, thus requiring a minimal knowledge about the system under analysis (it is sufficient to define the use-cases for the main functionalities). Aspects descend from the requirements, thus restoring an alignment with the implementation.

3.2 Aspectizable interfaces

In this section another aspect mining technique is described, based on the notion of the role which sometimes is attached to an OOP interface\(^5\).

3.2.1 Aspectizable interfaces definition

The principal decomposition of a program is typically apparent from the hierarchy of its classes. In fact, the inheritance relationship usually models the refinement of abstract into concrete entities, according to a model of the application domain.

In addition to extending the super-class, classes may implement interfaces. Interfaces provide alternative decompositions of the functionalities, according to a different, possibly orthogonal view. For this reason, interfaces often (though not always) do not belong to the principal decomposition of a system.

When a program is designed according to the OOP paradigm, the hierarchy of the classes reflects the (principal) decomposition of data structures and functions into smaller, composable units. In such a decomposition, the interfaces play a twofold role:

1. An interface may collect abstract properties of the principal decomposition, shared by the classes implementing it.

\(^5\)Not to be confused with graphical user interfaces (GUI).
2. An interface may collect transversal properties, that crosscut the principal decomposition. Such properties recur across multiple unrelated classes, instead of being confined within a single, cohesive group of classes.

We call the latter an aspectizable interface.

Figure 3.5: Class diagram for a portion of the collection framework from the Java standard library.

If we consider the organization of the collection framework in the Java standard library (see Figure 3.5, for a portion of the corresponding class diagram), the interface Collection is a good example of the first kind, while Serializable belongs to the second case. In fact, the interface Collection is used to describe the container role that is played by classes from different sub-hierarchies, thus clearly contributing to the definition of the main organization of the library into smaller modularization units. On
the other hand, the interface `Serializable` is not specific to the organization of the collection framework into sub-units, being rather a transversal property of the classes in this package (and also of classes in other packages).

Let us consider persistence (interface `Serializable`). The code fragments implementing this interface are spread across several classes (scattering). Moreover, this code requires access to information about each entity to be stored persistently (tangling). On the other side, if we consider the persistence functionality from a logical point of view, it clearly does not belong to the principal decomposition of the application. Rather, it is a transversal computation that has to be superimposed. In other words, it is an *aspect* of this application.

The reasons for considering the `Serializable` interface as an aspect hold for many of the interfaces that are usually implemented by the classes in a Java application. Another example is the interface `Cloneable`, which allows duplicating existing objects. When necessary, this functionality is added. However, it typically does not take part in the main decomposition of the system.

Another example of usage of interfaces is associated with the implementation of design patterns [13]. Roles are superimposed to the modules in a subsystem by making them realize given interfaces. For example, `Observable` and `Observer` interfaces can be introduced to specify the roles played by different classes organized according to the Model View Controller (MVC) pattern. Since role superimposition is typically orthogonal to the principal decomposition, this usage of the interfaces usually falls into the category of the aspectizable ones.

Since the behavior captured by an aspectizable interface is shared by classes spread across different hierarchies, its OOP implementation consists of code fragments that are part of several classes (scattering). Moreover,
the implementation requires intimate knowledge of the classes possessing such a behavior, giving raise to tangled code. When a behavior implemented in a software system is scattered (non local implementation) and tangled (highly coupled) with the remaining code, the maintainability is expected to be affected negatively. Changes of such a behavior cannot be implemented locally (due to scattering) and ripple effects are hard to predict (due to tangling). A system property with these two features is clearly a crosscutting concern that AOP can address.

It should be noticed that the issues related to the implementation of the aspectizable interfaces generalize to all programming languages that support multiple object substitutability, i.e., the possibility for an object to play multiple roles in different contexts. In fact, one of such roles is typically dominant, the others being crosscutting with respect to the principal decomposition. For example, in the C++ language, which supports multiple inheritance, the principal decomposition is usually a slice of the class hierarchy, while some of the super-classes are better regarded as the declaration (possibly including a default implementation) of crosscutting properties. The relationship between aspects and multiple inheritance/roles is clarified in two papers by Hanenberg and Unland [34, 35]. Role-based refactoring of crosscutting concerns is described in a paper by Hannemann et al. [38].

Interface implementation seems a good starting point for the identification of candidate aspects in an existing system. It involves the introduction of methods in a class – those required by the interface. Moreover, sometimes a proper (e.g., efficient) behavior of the interface methods require that additional attributes (e.g., a cache) be inserted into the given class and that inner classes be defined. When an interface implementation is recognized as a crosscutting concern, it can be migrated to an aspect.

In the refactored system, where the interface implementation becomes
an aspect, software maintenance is expected to be simplified. In fact, program understanding becomes easier, since the code of the classes represents only the principal logical decomposition, not mixed with other views. Moreover, each crosscutting concern (e.g., serializability, cloneability, etc.) is located in a separate modular unit.

3.2.2 Aspectizable interfaces identification

The purpose of aspect mining is the identification of those interfaces, implemented in a given class, which can be regarded as crosscutting concerns and can be subjected to refactoring in order to aspectize them.

While a fully automated aspect identification process is not feasible, because of the fuzzy notion of principal decomposition and of the level of subjectivity involved, it is possible to define a set of indicators that hint a high likelihood that a given interface implementation represents an aspect. The result of computing such indicators must then be interpreted by a human, making the final decision.

We have defined the following set of aspect mining indicators specifically for the migration of interface implementations to aspects. The implementation of an interface is marked as a candidate aspect when:

**External package** The interface implemented in a class belongs to a package different from that of the given class.

**String matching** The name of the interface implemented in a class matches a user defined pattern (e.g. ".*able").

**Clustering** When methods are clustered according to the call relationship, interface methods are not grouped together with other (non-interface) class methods.
Unpluggability The methods of the interface implemented in a class can be unplugged from the given class, since they are not invoked by other methods of the same class.

The first criterion assumes that interfaces in the principal decomposition are declared in the same package as the classes under analysis. The second criterion was derived from the aspect mining methods based on string matching [36]. The example of regular expression matched against each interface name consists of an arbitrary prefix (".*") followed by the suffix "able". Such a suffix typically indicates an additional property of the given class, orthogonal to its main properties (e.g., Serializable, Cloneable).

The rationale behind the last two aspect mining criteria is that methods inserted into a class to implement an interface which is not in the principal decomposition are loosely coupled with the other methods of the class, since they do not contribute to the main functionalities of the class. The Unpluggability criterion makes the (strong) hypothesis that no method calls any one of the interface methods, except for the interface methods themselves. A weaker hypothesis, made in the Clustering criterion, is that the call relationship identifies subgroups of cohesive methods and that interface methods are never in a group including also non-interface methods. In other words, calls from non-interface methods are admitted, as long as calling methods are not highly coupled with the called interface methods.

The clustering method used to determine groups (clusters) of highly connected methods has been defined along the lines given in [46]. A metric of modularization quality, accounting for the difference between cohesion and coupling, is maximized by means of a proper combinatorial optimization heuristics (we used hill climbing).

Once an interface is classified as an aspect candidate, not only its methods are considered part of the crosscutting concern, but also all fields and inner classes functional to the interface implementation. In order to iden-
identify which fields and inner classes should be assigned to the aspect being migrated, the following, simple criterion was adopted:

*Fields and inner classes are migrated to the aspect associated with a given interface if they are used inside, but not outside the interface methods.*

### 3.2.3 Tool implementation

Automated support to *aspect mining* requires the execution of some code analyses on the input program. Information on the interface names and packages, and on the call relationships between methods has to be recovered, in order for the proposed aspect mining techniques to work. The static analyzer that performs this job was written in the source code transformation and analysis tool TXL [22]. A few small Java programs have been developed to implement the aspect mining checks described previously, based upon the output produced by the TXL analyzer.

**Call relationship module** A TXL program has been implemented to analyze the source code and extract all the call relationships among methods. Only internal calls are considered, so all the calls to methods defined in library classes are ignored here.

**Interface methods module** Another TXL program is used to collect all the interfaces implemented by the current class. All the class methods required by each interface are also detected.

**Hill climbing module** The clustering algorithm has been implemented within a Java tool. Since the problem space is too wide, an heuristics has been used to find a sub-optimal solution, hill climbing. The algorithm starts by computing the fitness function (modularization quality, see previous section) on a random solution, then the fitness
function is evaluated on all the neighbour solutions. A random solution in chosen among all the neighbour whose fitness is better than the current one and it becomes the current solution, so the fitness improves. The same step re-applies until no more improvements can be reached. At this point the current solution represents the outcome of the algorithm.

Having this framework available, implementing the four mining techniques is straightforward:

**External package** the *Interface method* module provides the complete interface names. Their packages are compared with the current class package.

**String matching** Interface names, provided by the *Interface method* module, are matched with the regular expression ".*able".

**Clustering** The relationship extracted by the *Call relationship* module are subjected to the *Hill climbing* module. Methods from the *Interface method* module are detected in the solution of the previous step, in order to verify whether interface methods belong to a cluster which contains also non-interface methods.

**Unpluggability** informations from the *Call relationship* and the *Interface method* modules are analyzed to verify whether the interface methods participate in a call relationship with non-interface methods.

### 3.2.4 Contribution to the state of the art

As most of the aspect mining approaches in the state of the art (see Section 2.3), the identification of the aspectizable interfaces relies on the static analysis of the class structure.
Similarly to our clustering heuristics, two other works [56, 39] used clustering to reveal aspect candidates. While in the former [56] the distance information used to cluster together methods is based on their name similarity, in the latter [39] method call information is considered. However, the authors of [39] try to identify methods that are frequently called together, based on the assumption that the recurring call pattern represents a candidate crosscutting concern. Instead, in our approach clustering reveals the separation between the aspectizable interface realization and the rest of the class.

Other authors proposed heuristics, such as unique method [32] and call fan-in [48], to identify aspect candidates. Those approaches are similar to ours, in that they are based on the assumption that a crosscutting concern verifies specific properties which are captured by the corresponding heuristics. But, while existing heuristics are applicable to almost every method in the system, our heuristics apply only to those methods that contribute to an interface realization. Thus, the search space is much more restricted in our case.

Like in our string matching heuristics, the assumption that naming convention is important in aspect mining has been previously investigated, using natural language processing [57] and formal concept analysis [66]. Instead of analyzing a big portion of the whole program as in the cited approaches, our heuristics focuses on a very localized segment, the interface name. In fact, we try to reveal whether the developer meant to attach a particular semantics to the interface while designing it, capturing the case when such semantics corresponds to the notion of the aspectizable interfaces.
Chapter 4

Refactoring

In this chapter the problem of how to refactor a crosscutting concern into an aspect is addressed, assuming that aspect mining has already been applied and crosscutting concerns have already been identified. The first section deals with the refactoring of the aspectizable interfaces. The refactoring of an aspectizable interface consists of modifying all the classes that realize such interface, where all the interface methods are moved from the classes to an aspect. In the second section, all remaining references to the crosscutting functionality are removed from the principal decomposition. After removing methods from the base code, all the calls to these methods are taken off and replaced by proper aspect pointcuts and advices, in order to make the base code oblivious of the crosscutting concern.

4.1 Aspectizable interfaces

4.1.1 Refactoring process description

Migration of an aspectizable interface to an aspect involves two main, high-level code transformations (refactorings, [26]):

1. *Move properties to aspect*: properties (attributes, methods, inner classes) are modularized in the aspect, that introduces them into the affected
classes.

2. **Remove references to properties**: execution points referencing aspectizable properties are moved into the aspect code (called *advice code*) triggered by the pointcuts.

In the case of the aspectizable interfaces, the first transformation is the most important one, since the methods in the interface implementations are seldom referenced by methods in the principal decomposition.

Figure 4.1 shows the mechanics of the first refactoring. The overall transformation can be described in terms of three simpler refactoring steps, applied repeatedly:

- **Move method to aspect.**
- **Move field to aspect.**
- **Move inner class to aspect.**

These three (atomic) refactorings consist of removing a method (resp. field or inner class) from a given class and adding it to an aspect, where it becomes an introduction.
In Figure 4.1, class A implements the interface I by defining the body of methods if1, if2, if3, the class field x is used only inside if1, if3, and the inner class AA is used only inside if2, if3. Moving the interface implementation to a new aspect IA consists of applying the three steps above respectively to if1, if2, if3, to x, and to AA.

The result (see Figure 4.1, right) is a thinner class A, with only 1 field (y) and two methods (f4, f5), which depends (dashed edge) on the aspect IA for the implementation of the interface I (see tag over the realization relationship). Inclusion of the inner class AA is also dependent on the new aspect IA (tag over nesting relationship).

Inheritance of interfaces is handled by computing the union of all super-interface methods (flattening). In fact, when a class implements an interface it must also implement all the super-interface methods. Thus, the methods declared in the super-interfaces are also migrated to the aspect being constructed.

When an interface migrated to an aspect is implemented by several classes in the system under analysis, additional advantages can be potentially obtained from the separation of the crosscutting concern represented by the interface. In fact, if the different implementations of the interface share some computations, it becomes possible to factor them out into a common super-aspect.

Figure 4.2 shows the output generated by the refactoring, when it is applied to the example depicted in Figure 4.1 (inner class excluded). The responsibility of implementing the interface I is taken out of class A and is assigned to the aspect IA, so as to leave only principal decomposition methods inside A (assuming I is an aspectizable interface). Thus, one aspect is generated for each implementation of a given aspectizable interface.

The main benefit of the refactoring described above is that in the principal decomposition, which does not include IA, several entities and class
properties are removed. This reduces the program comprehension effort required to understand the principal system’s organization. Specifically, in the principal decomposition the interface I and the inner class AA disappear, and so do the methods if1, if2, if3 and the attribute x, no longer listed among the properties of class A.

The role of aspect IA is clarified in a secondary decomposition, focused on the implementation of the interface I. This alternative view, which basically consists of the right part of Figure 4.1, indicates that weaving class A with aspect IA results in class A implementing interface I and containing the inner class AA. Separation of this secondary concern from the principal decomposition and its localization inside an aspect is also expected to affect positively the understandability of the application.

Abstracting aspects

When an interface migrated to an aspect is implemented by several classes in the system under analysis, additional advantages can be potentially
obtained from the separation of the crosscutting concern represented by the interface. In fact, if the different implementations of the interface share some computations, it becomes possible to factor them out into a super-aspect.

Let us consider three classes $A$, $B$, and $C$, implementing the interface $I$ with interface methods $if1$, $if2$, $if3$. After applying the refactoring in Figure 4.1, three new aspects $IA$, $IB$, $IC$ are created to separate the concern of interface implementation from the original classes. If the method introductions in the three aspects perform the same sub-computation, it is possible to separate it from the class-specific computations and move it to a new method ($af1$ for the common sub-computation in $A.if1$, $B.if1$, $C.if1$, and similarly $af2$ and $af3$). It should be noted that common sub-computations are quite likely to occur in practice in different implementations of the same interface methods, because (without the aspects) such implementations are spread across different classes, where factorization of the common sub-computations in a unique modular unit may be difficult to realize.
The common computations $af_1$, $af_2$, $af_3$ can be inserted into the classes A, B, C by a super-aspect $II$, extended by the three aspects $IA$, $IB$, $IC$ (see Figure 4.3). In this super-aspect, a generic introduction is performed, using the type variable $T$. Sub-aspects bind the type variable $T$ to the classes they modify, so as to obtain the necessary introductions.

Identification and factorization of common computations into super-aspect introductions is a human-intensive activity, in that it involves lots of domain and application knowledge, and non-trivial decision making. However, partial automation can be achieved by exploiting clone detection techniques [3]. The presence of a cloned code fragment inside interface methods introduced in different classes indicates the possibility of abstracting the related aspects and factoring out the cloned computation.

When common computations are identified in different aspects and abstracted into a super-aspect, the modularization power of AOP becomes even more evident. In fact, while initially the same interface is implemented in different classes by means of replicated code fragments, in the refactored system the commonalities among the different implementations are localized in a single module (no more scattering). In other words, the presence of the same crosscutting sub-computation is represented explicitly in the new modular unit being created, the super-aspect.

### 4.1.2 Tool support

The refactoring toolkit consists of the modules depicted in Figure 4.4, which are written in TXL. The main module, shown at the top, implements the actual code transformation. Given an input source (e.g., A.java) and an interface to be migrated to an aspect (e.g., I.java), the TXL module $UNPLUG$ produces a new source file ($A'$.java), in which the interface implementation is absent, and an aspect (file $IA$.java), which introduces the interface implementation into the original class. The aspect is also
responsible for declaring that the class implements the interface (declare parents construct).

The other modules in Figure 4.4 are accessory modules that give specific information about interface aspectization. Module UnpluggableFields produces the list of fields that are accessed only from interface methods. Similarly, UnpluggableInnerClasses gives the inner classes not accessed outside the aspect being constructed. Finally, InterfaceIsUnpluggable gives information about the role of the aspect under construction. In case this module produces an output yes, the aspect will be regarded as an optional class feature, to be weaved with the class only upon request from the client. Otherwise, the aspect has to be considered an essential feature, because the methods it introduces are used also by other class methods.

When the aspects generated for the interfaces implemented in a class are optional, clients of the given class have the freedom to decide which
interfaces the class should implement. According to their needs, they can select which aspects to weave with the class and which other aspects to exclude. In the overall architecture, the realized interfaces can thus be configured on demand.

In the development of the toolkit described above, some limitations of the current version of AspectJ have been identified. They prevented us from obtaining an implementation completely consistent with the approach presented in the previous sections. This is the list of the main problems encountered and of the workarounds we have adopted:

1. Private methods cannot be invoked (while private fields can be accessed) from methods introduced by an aspect.

2. Inner classes cannot be introduced by an aspect.

3. Genericity is currently not supported.

The workaround used for (1) was changing the visibility of methods where required. Problem (2) was resolved by bringing the inner class at top level with package-protected visibility. For (3), when possible, shared sub-computations introduced by super-aspects were introduced in a common super-class. Otherwise, aspect methods instead of introductions have been used.

### 4.2 Pointcut extraction

The pointcut extraction algorithm has two steps. The first, which does the bulk of the work, uses refactoring to create the initial pointcut and advice instances necessary to aspectize the OO code selected for migration. The second step combines these instances into the final pointcuts and advices. This section first describes the refactorings and then the merging. The
identification of aspect candidates (aspect mining) and the mark-up of the code regions that represent instances of these candidates to be refactored are assumed to have been completed before the extraction begins.

The regions of code marked for refactoring fall into one of three cases:

1. whole methods or fields,
2. calls to methods, or
3. statement sequences (blocks of code).

Extraction of whole methods or fields (Case 1) requires moving the method or field from a class to an aspect, where it is turned into an AspectJ introduction. It has been described in details in the previous section with reference to the aspectizable interfaces. Case 3 (statement sequence) can be easily reduced to Case 2 by applying the well-known OO transformation *Extract Method*. For this reason, this section focuses on Case 2. In fact, a sequence of arbitrary statements to be aspectized can be handled by extracting them into a separate method, aspectizing the call to such method and (if possible and desirable) inlining the method body into the advice. In this way, the same mechanics applied to calls can be used for arbitrary statements.

Among the high number of refactorings for migration from objects to aspects that have been proposed [9, 21, 33, 38, 49, 68], this section focuses on a small subset, selected using a mix of a bottom-up and of a top-down approach. A-priori knowledge about the constructs provided by most aspect languages/frameworks (including AspectJ) and about the kinds of crosscutting concerns that are amenable for implementation through aspects guided us. Furthermore, here we consider only refactorings whose mechanics can be fully automated. In fact, one of the aims of the present thesis is to assess the ability of this small list of refactorings to cover most
of the cases encountered in practice. The following six refactorings from objects to aspects have been included:

- **Extract Beginning/End of Method/Handler:** The marked code is at the beginning/end of the enclosing method body or of one of the method’s exception handling blocks.

- **Extract Before/After Call:** The marked code is always before or after a method call.

- **Extract Conditional:** A conditional statement controls the execution of the marked code.

- **Pre Return:** The marked code is just before the return statement.

- **Extract Wrapper:** The marked code is part of a wrapper pattern, in which the wrapper code is to be aspectized.

- **Extract Exception Handling:** The marked code is a whole exception handling block.

When none of the refactorings above apply to a marked code fragment, OO transformation is resorted to (also called OO refactorings [26]) in order to make one or more of the refactorings above applicable. Among the possible OO transformations, the following are regarded as the two most important ones: *Statement Reordering* and *Extract Method*. Both can be fully automated (the latter is available in most transformation environments, while the former requires some non trivial dependency analysis to ensure semantics preservation). Hereafter *transformations* are OO to OO and refactorings are OO to AOP.

*Statement Reordering* allows the order of two statements to be exchanged. In its simplest form, this requires that the defined and referenced variables of the two statements do not overlap. When some overlap
does occur between defined and referenced variables, it may be possible to make this transformation applicable by introducing fresh local variables that store a value that must be preserved.

The second OO transformation, *Extract Method*, allows a sequence of statements to be extracted to a separate method [26]. Method arguments might be required if local variables or parameters of the original method are referenced in the marked statement block. This transformation makes it possible to aspectize virtually *any* marked block of code. However, it impacts the structure of the base code quite deeply, so it is used as sparingly as possible and as a “last resort”.

The remainder of this section provides details on the two steps of the extraction algorithm: first it describes the six refactorings from objects to aspect advice instances. Their description is based on a set of examples, which cover the most common cases. A formal presentation of the mechanics associated with each refactoring in the general case is given in the Appendix. The second step, which merges the advice instances to form the final aspect, is then described.

### 4.2.1 Extract Beginning/End of Method/Handler

This refactoring deals with the following case:

*The marked code is at the beginning/end of the enclosing method body or of one of the method’s exception handling blocks.*

Fig. 4.5 shows the result of applying the refactoring *Extract End of Method* to a small code fragment. The call to method `g` is removed from the body of `f`. A new aspect, named `B`, is created to intercept the execution of `f` and insert a call to `g` at the end. The target of the call (*this*) is accessible within the aspect advice because of the advice parameter *This*, which is bound to *this* by pointcut `p`. The actual parameter of the call
4.2. POINTCUT EXTRACTION

class A {
    int x = 0;
    void g(int y) { }
    void f(int y) {
        x += y;
        this.g(y);
    }
}

class B {
    pointcut p(A This, int y):
        execution (void A.f(int)) &&
        this(This) && args(y);

    after (A This, int y): p(This, y)
    { This.g(y); }
}

Figure 4.5: Example of refactoring: Extract End of Method.

(y) is made accessible (exposed) within the aspect advice because of the advice parameter y, which is bound to the method parameter y by the args construct used in pointcut p.

The variant of this refactoring with the call to be aspectized at the beginning of the method requires simply changing the after advice into a before advice. The variant with the call in the exception handling block requires simply concatenating (through &&) a handler primitive pointcut at the end of p. When the re-application of this refactoring produces a pointcut already generated before (e.g., Extract End of Method applied twice to the same method), the pointcut and the associated advice are not duplicated. The marked code is inserted at the beginning/end (resp. for Extract End/Beginning) of the previously generated advice body. If a method has multiple exit points, the same marked code fragment must appear before each of them in order for this refactoring to be applicable.
4.2.2 Extract Before/After Call

This refactoring deals with the following case:

*The marked code is always before or after another call.*

```java
class A {
    int x = 0;
    void g(D d) {}
    void f(C c) {
        D d = c.dd();
        c.h();
        this.g(d);
        if (x > 0)
            x = 0;
    }
}
```

```java
aspect B
[percflow (execution(void A.f(C)))] {
    D d;
    pointcut p1(A This):
        withincode (void A.f(C)) &&
        this(This) && call (void C.h());
    after (A This): p1(This)
        { This.g(d); }
    pointcut p2():
        withincode (void A.f(C)) &&
        call (D C.dd());
    D around(): p2() {
        d = proceed();
        return d;
    }
}
```

Figure 4.6: Example of refactoring: Extract After Call.

Fig. 4.6 shows an application of the refactoring *Extract After Call*. In the aspect B, the pointcut p1 intercepts the call to h that occurs within the
execution of method \( f \). After-advice reintroduces the call to \( g \) after each call to \( h \).

Pointcut \( p2 \), the related advice and the \texttt{percflow} clause are not necessarily part of this refactoring. They need to be included only if a local variable of the intercepted method has to be exposed in the aspect (which can occur in any of the six refactorings presented herein). Often, successive refactorings make the context exposing pointcut (\( p2 \)) unnecessary, since the local variable becomes completely confined within the aspect code. In such cases, the aspect can be simplified and the context exposure code can be removed altogether.

Since the call to \( g \) includes the local variable \( d \) as a parameter, a copy of the local variable is created inside the aspect. This is achieved by intercepting every assignment to the local variable (pointcut \( p2 \)) and copying the value into the aspect variable before returning it (around-advice associated with \( p2 \)). In order to create a fresh new copy of the variable in the aspect each time the control flow reaches the execution of the method containing the local variable, the aspect must be instantiated for each control flow (\texttt{percflow} clause) in which the method is executed. It should be noticed that this is the only context exposure mechanism that cannot be fully automated and requires human intervention.

In cases when the local variable is no longer used in the base code, it can be turned into an aspect variable. This happens typically when the local variable is itself part of the concern being aspectized. With reference to the example in Fig. 4.6, let us consider what happens if the first statement of \( f \) is also to be aspectized. Moving it to the aspect \( B \) is a straightforward application of the refactoring \textit{Extract Beginning of Method} described before. Once this second refactoring has been applied, the set of join points intercepted by pointcut \( p2 \) becomes empty (this is easily checked by the AspectJ compiler). Correspondingly, there is no need for the code fragments
within dashed boxes in Fig. 4.6, which can be removed from the aspect. The resulting aspect code is much clearer and closer to the code that would be written manually. Thus, although the context exposure associated with local variables might be complex, it is an acceptable intermediate step if the local variable is part of the concern and is completely moved to the target aspect by successive refactorings.

In the presence of multiple calls to the intercepted method (e.g., h in Fig. 4.6) the refactoring Extract Before/After Call is applicable only if each call is preceded/followed by the call to be aspectized (g in Fig. 4.6). If this is not the case, the refactoring is not applicable and usage of some enabling transformation (such as Extract Method) should be considered.

### 4.2.3 Extract Conditional

This refactoring deals with the following case:

*A conditional statement controls the execution of the marked code.*

Fig. 4.7 shows an example of this refactoring. The conditional statement if (b) is considered to be part of the aspect, in that it determines the execution of the call being aspectized (in Fig. 4.7 the call to g). Thus, it becomes a dynamically checked condition incorporated into the aspect’s pointcut (using the AspectJ syntax if (This.b)). For the execution to be intercepted by pointcut p, the condition This.b must be true; in which case, the new body of method f is replaced by the call to g, as specified in the around-advice.

Two variants of Extract Conditional are worth mentioning. Firstly, if the "x++;" were not under the control of condition b (placing it at the top-level in f) it would be sufficient to add a proceed statement at the end of the around-advice to ensure that it is always executed (both when the advice is triggered and when the execution flows normally). Secondly,
4.2. POINTCUT EXTRACTION

class A {
    boolean b = true;
    int x = 0;
    void f() {
        if (b) {
            this.g();
        } else {
            x++;
        }
    }
}

aspect B {
    pointcut p(A This):
        execution (void A.f()) &&
        this(This) && if(This.b);
    void around (A This): p(This)
        { This.g(); } }

Figure 4.7: Example of refactoring: Extract Conditional.

if method g is in the else-part of the conditional statement, it is sufficient to use if (!This.b) instead of if (This.b) in the pointcut.

4.2.4 Pre Return

This refactoring deals with the following case:

The marked code is just before the return statement.

Fig. 4.8 shows an example of Pre Return. The call to g is moved from the method body to the around-advice. The advice code contains a proceed invocation, which triggers the execution of the intercepted method f. Its return value is stored into a temporary variable (y) and returned after the invocation of the aspectized statement (i.e., the call to method g), which changes its value.
This refactoring is a variant of Extract End of Method that occurs whenever the code to be aspectized is at the end of a value-returning method. The main difference is the usage of an around-advice, instead of an after-advice, which is required whenever the returned value is affected by the aspectized statement. Preservation of the semantics can be achieved by: (1) executing the original method (via proceed); (2) executing the aspectized call (in the advice); and (3) returning the final value. An after-advice would swap the execution order of (2) and (3).

4.2.5 Extract Wrapper

This refactoring deals with the following case:

The marked code is part of a wrapper pattern, in which the wrapper code is to be aspectized.
Fig. 4.9 shows an example of *Extract Wrapper*. The object \( x \) is wrapped into \( y \) before being used as the actual parameter of a call to \( g \) (see also Gamma et al.’s pattern [27]). In order to move the creation of the wrapper object (underlined statement in method \( f \)) to the aspect, the un-wrapped object \( x \) is used in the refactored code for the method \( f \) as the actual parameter of the call to method \( g \). Such a call is intercepted by the pointcut \( p \), which exposes its argument. The associated around-advice uses this argument, which is known to belong to class \( X \), to create the wrapper object \( y \). This object is passed to method \( g \) by restoring the original method invocation (\texttt{proceed} construct) with a new argument.

Similar to *Call Before Method*, *Extract Wrapper* is applicable only if the body of \( f \) contains just one call to \( g \), or multiple calls all of which participate
in the same wrapper pattern. If this is not the case, application of this refactoring in an alternative form can be considered. The pointcut \( p \) may intercept the creation of the object \( x \), instead of the call to \( g \), by means of the pointcut designator "call(X.new())". By exposing the target of the call to the constructor (target construct in AspectJ), the un-wrapped object \( x \) can be made available within the around-advice, which will contain exactly the same code as the around-advice shown in Fig. 4.9.

4.2.6 Extract Exception Handling

This refactoring deals with the following case:

The marked code is a whole exception handling block.

```java
class A {
    void f() {
        try {
            g();
        } catch (E e) {
            exit(2, e);
        }
    }
}
import org.aspectj.lang.SoftException;
aspect B {
    declare soft: E:
        (call (* *(..) throws E) ||
        || call (*.new(..) throws E) ) &&
        withincode (void A.f());
after() throwing (SoftException e):
    execution (A.f()) {
        exit(2, (E)e.getWrappedThrowable());
    }
}
```

Figure 4.10: Example of refactoring: Extract Exception Handling.
As an example, consider the refactoring of the exception handling code marked in Fig. 4.10, so as to move it to an aspect. The remaining code should become oblivious to any exception of type \( E \) raised in the body of method \( A.f \). This is achieved by means of the AspectJ construct `declare soft`, used to soften all exceptions of type \( E \) raised within the code of \( A.f \) (see Fig. 4.10). The newly created aspect \( B \) can intercept the exceptions of type `SoftException` raised during the execution of \( A.f \). Since these are produced by the softening of exception \( E \) in aspect \( B \), it is possible to obtain the original exception (through invocation of `getWrappedThrowable`) and to restore the original exception handling computation, by means of an after-advice. This refactoring was suggested by Laddad [45].

### 4.2.7 Pointcut Abstraction

The above refactorings extract each instance (code fragment) of an aspect marked in the code. The final step, *pointcut abstraction*, merges the separate instances into a single advice whenever possible.

The merge is primarily a syntactic combination of the pointcuts and advices for the extracted aspect instances. When the advice bodies are exactly the same, the combination requires just determining the union of the pointcuts produced by refactoring. For example, a pointcut clause may describe the call to a particular method being intercepted. The remaining clauses describe the context in which the advice is to be woven. A disjunction of these clauses is formed and added to the pointcut specifying the call.

The merge of the advices is often trivial. As seen below, the predominant case is for the instances of a given aspect to have the same advice. Minor differences can be accommodated by the inclusion of conditionals. One exception comes up when the exposed context involves different types. For example, if some logging code was called from method \( m1 \) of class \( C1 \) and
from method m2 of class C2, then the pointcut instances would be:

\[
\text{before(C1 This) : log_pointcut (This)} \quad \cdots \\
\text{before(C2 This) : log_pointcut (This)} \quad \cdots
\]

Here it is necessary to replace C1 and C2 with a unique type, used for the parameter This. In this case, standard practice is to create a new interface for the advice and use it in place of C1 and C2. The aspect declares that C1 and C2 implement such an interface.

### 4.2.8 Iterative Refactoring Process

The migration process makes the assumption that aspect mining has already been performed on the system. The result of mining is a marked source code, where all the code fragments that have been identified as part of a crosscutting concern are surrounded by block comments containing special annotations.
Figure 4.11 shows an overview of the migration process. The process consists of a loop over all the marked portions of code. In turn, each marked segment of code is analyzed in isolation and, with the involvement of the user, moved to an aspect. The iteration goes on until no more marked statements remain in the base code.

A process iteration consists of several steps, the first of which is *Discovery*. During this step, the tool analyzes the current marked code and compares it with the refactoring patterns. A match occurs when the corresponding refactoring is applicable to the current code fragment, so it can be used to transform it. The output of *Discovery* is the list of applicable refactorings. Items in the list are ranked top-down, with the best refactoring at the top. A relative scale of priorities among the refactorings was defined, based on the impact that each refactoring has on the original code and the complexity and generality of the aspect code that is generated.

*Extract-conditional*, *extract-wrapper* and *extract-exception-handling* are not included in the priority scale, because they are bound to a very specific pattern. When the code matches one of them, no other pattern can match, so they are the only possible choice, thus there exists no priority relation. The highest rank in the priority scale is taken by *extract-beginning-of-method* and *extract-end-of-method*. Their refactoring mechanics define the generation of a very strong pointcut, because it relies only on a single method execution. This is a very robust pointcut that could keep its validity through future maintenance activities that will involve a base code change.

The next priority level is for *extract-pre-return*, because it is quite similar to *extract-end-of-method*, but it produces a slightly more complicated aspect code. In fact, it has to store the intermediate return value that will be returned after the execution of the concern code.

The lower priority is for *extract-after-call* and *extract-before-call*, because
they are applicable only if the method call used to intercept the original code is unique in the whole method body. This same condition must remain true also during the evolution of the system source code. If in a future maintenance change another call to the same method is introduced, the aspect will intercept more join point than required, potentially producing an inconsistent behavior. The code generated with these refactorings is quite fragile, thus the adoption should be limited.

In the second process step, Selection, the user chooses which refactoring to apply among the applicable ones, either following or ignoring the assigned priority. The tool (which will be described in the next subsection) allows the user to evaluate the alternatives by looking at a preview of the resulting source code for each refactoring, before actually applying it. The user can also decide to leave the current portion of code unchanged and go on with the next portion, deferring the transformation to a future iteration. Of course, this is mandatory when no refactoring pattern matches the current portion of code.

In the Transformation step, enabling transformations are used to change the current code fragment in such a way that it matches a refactoring pattern previously un-matched, thus making the corresponding refactoring applicable. These enabling transformations can be well-known object oriented transformations, such as *Extract Method* or *Statement Reordering* or ad-hoc (manual) transformations.

If one of the proposed refactorings is considered appropriate, the user simply selects it for the next process step, Refactoring. In this fully automatic step, the marked code is removed from the class and is moved to an aspect advice. A proper pointcut is generated to intercept the join point in the modified base code, according to the mechanics of the selected refactoring (as detailed in Appendix A).
4.2.9 Tool Implementation

AOP-Migrator has been implemented as a plug-in for the Eclipse\textsuperscript{1} framework, on top of the Java (JDT) and AspectJ (AJDT) plug-ins\textsuperscript{2}. AOP-Migrator uses several framework facilities to perform the code analyses required when evaluating the refactoring preconditions. Moreover, facilities already available in Eclipse (\textit{e.g.}, object oriented transformations, edit commands, previews) are inherited by the tool.

The run-time behavior of the tool fits the refactoring iterative process described above. When the refactoring engine starts, it recognizes the marked statements, that are surrounded by the block comments \texttt{/**@begin-aspect\*/} and \texttt{/**@end-aspect\*/}. The first marked piece of code is taken into account and is displayed in some dialog windows.

The interaction with the user starts with the dialog shown in Figure 4.12(a). The user is required to specify the target aspect (where the concern code is placed). The default choice is the aspect selected in the previous iteration.

After that, the wizard proposes the list of applicable refactorings ordered by priority (see Figure 4.12(b)). The \textit{No refactoring} item is always present. When the user selects a refactoring, the wizard shows the preview dialog (Figure 4.12(c)). In the preview dialog, the upper panel lists all the source files that are involved in the change. If a file is selected in the upper panel, the related detailed change is shown in the lower panel, where the source code before and after the refactoring is compared. Up to this point, the refactoring has not yet been applied and the data in the previous dialogs can still be modified, by navigating with the \textit{Back} and \textit{Next} buttons. The change is actually committed only when the user presses \textit{Finish}.

Some enabling OO transformations are already available in the Eclipse\textsuperscript{1}

\textsuperscript{1}http://www.eclipse.org.
CHAPTER 4. REFACTORING

4.2. POINTCUT EXTRACTION

Figure 4.12: Aspect selection dialog (a); Refactoring selection dialog (b); Preview of the resulting source code (c).

framework as defined in the JDT plug-in (e.g., Extract Method and Promote Local Variable to Field). Other transformations, not yet implemented, are performed manually.
4.3. CONTRIBUTION TO STATE OF THE ART

In its current version, AOP-Migrator checks the applicability conditions of the refactorings only at the syntactic level. Further human judgment may be necessary in specific cases. As with OO refactoring [26], it is highly advisable that each small code transformation step be followed by regression testing, via automated execution of unit tests. Although not ensuring semantics preservation, regression testing increases the confidence in the correctness of the transformation.

4.3 Contribution to the state of the art

The works most relevant to the approach presented in this chapter are those by Marin [47], Hanenberg et al. [33], Monteiro and Fernandes [49], Laddad [45], Gybels and Kellens [31] and Tourwe et al. [65]. Marin manually refactored the Undo concern present in JHotDraw from Java to AspectJ. The primary difference between this work and that which is reported in the present chapter lies in the level of automation. Marin’s goal was to (manually) understand the degree of tangling between the Undo concern and the base code. Marin applies preliminary OO transformations to reduce tangling, so as to produce code which is easier to migrate in much the same way we have done.

Hanenberg’s work [33] deals with the aspect oriented re-definition of popular OO transformations [26]. This work, and the work by Monteiro and Fernandes [49], consider refactorings to migrate from OO to AOSD and refactorings that apply to aspect code. Among them, the Extract advice refactoring [33] (or Extract Fragment into Advice [49]) is automated by AOP-Migrator using the first five refactorings. Our contribution to the definition of the Extract (Fragment into) Advice refactoring consists of the precise definition of its mechanics in the five sub-cases considered by us. For each case, we provide details about the applicability conditions, the context
exposure constructs, the transformation rules, as well as the prioritization of the alternatives. Our sixth refactoring corresponds to Laddad’s *Extract Exception Handling* [45].

Gybels and Kellens [31], and Tourwe’s et al. [65] works use inductive logic programming to transform an extensional definition (enumerative) of pointcuts into an intensional one. These works are complementary to that reported in this chapter, as they focus on the problem of generalizing and abstracting the automatically produced pointcuts. This is definitely a desirable supplementary step in the overall process.

Additional examples of refactorings from objects to aspects are given with reference to the role played by the classes involved in the implementation of design patterns [38]. Behavior preservation conditions for aspect refactorings are specified in the work by Cole and Borba [21].

The main difference between the content of this chapter and the most closely related works in the literature [33, 49, 45, 47] is that both the aspectizable interfaces refactoring toolkit and AOP-Migrator focus on the automation of the refactoring process. Moreover, rather than attempting to cover the entire spectrum of possible refactorings, the aspectizable interfaces refactoring toolkit and AOP-Migrator consider only a limited set of refactorings (one by the former and six by the latter) that were believed to cover most of the instances found in real code (see Chapter 5). As a matter of fact, AOP-Migrator is the first publicly available object-to-aspect refactoring tool, developed as an Eclipse plug-in.
Chapter 5

Assessment

In this chapter, all the aspect migration steps presented in the previous chapters are assessed by means of a number of experiments. First of all, the capability of dynamic aspect mining to reveal crosscutting concerns is be evaluated in terms of precision and recall on a gold standard, consisting of a manually discovered crosscutting concern. Then, the same mining method is compared with two other existing techniques taken from the literature, and some considerations are drawn about strong and weak points of each approach. All these three techniques will be eventually combined into some brand new techniques, in order to take advantage of the respective strong points and to overcome their individual weaknesses. After that, aspect refactoring is assessed, by evaluating the completeness of the proposed transformations in migrating some object oriented applications to aspects. Our approach to aspectizable interfaces is then assessed both in terms of its ability to identify them and to change their object oriented realizations into aspects. An empirical study was used to measure the improved quality of the final AOP code with respect to the original OOP code.
5.1 Assessment Subjects

In the assessment of the migration steps, a common suite of software systems was used. They are open source projects taken from the Internet. They range from component oriented classes to full applications with intermediate cases.

Software systems accounting for more than half a million lines of code have been considered in this assessment. Table 5.1 shows some dimensional data, together with the techniques with which they are used. The range of their application domain is quite wide. The size of these applications is typical of medium (non trivial) Java systems. The number of Lines of Code (LoC) was measured on all source files for the application classes, test classes excluded. It includes comments and blank lines. Classes in the standard JDK library distributed by Sun (http://java.sun.com/) were taken from all packages nested inside java, including all sub-packages. JHotDraw (http://www.jhotdraw.org/) is a two-dimensional graphics framework for drawing editors, based on Erich Gamma’s JHotDraw. It includes a standalone application to draw geometrical shapes of different kinds. FreeTTS (http://freetts.sourceforge.net/) is a speech synthesis system written in the Java programming language. FreeTTS was developed by Sun Microsystems, based upon Flite, a small, fast, run-time speech synthesis engine, which in turn is based upon University of Edinburgh’s Festival Speech Synthesis System and Carnegie Mellon University’s FestVox project. JGraph (http://www.jgraph.com/) is a graph visualization library, including several graphical and algorithmic functionalities. We used the source code of version 1.4.0 of JDK, version 5.4b of JHotDraw, version 1.2beta of FreeTTS, and version 3.1 of JGraph. PetStore is a J2EE e-business application supporting order, sale, and shipment of products purchased on the Web. JSpider is a Web robot able to download a target Web site and to
check some internal server and link errors. JAccounting is a Web-based business accounting system, which includes support to invoice and account management, as well as Web access.

In one particular experiment, only two sub-packages of the first application (JDK classes) were used as case studies. These two sub-packages have been treated as separate systems, because of their specific individual properties.

The first sub-package (java.util) provides general purpose data structures, such as lists, sets, trees and hash tables. They offer a very popular programming framework, used by several software developers, who do not have to re-write their own implementation of these common data structures. They are used by classes in other library packages as well. Because of the general purpose, all the classes in this package implement many functionalities that could be useful in software development (for example serialization, cloneability, etc.), although these may not be required in all uses. Moreover, they are typically not essential in defining the meaning of each class, while being useful to enrich it. Thus, we expect that several of them can be regarded as crosscutting concerns.

### Table 5.1: Java programs, case studies of the present assessment.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (LoC)</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK classes</td>
<td>382,533</td>
<td>Aspectizable interfaces</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>40,022</td>
<td>Dynamic aspect mining, Mining comparison</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Aspectizable interfaces, Pointcut extraction</td>
</tr>
<tr>
<td>FreeTTS</td>
<td>31,009</td>
<td>Aspectizable interfaces</td>
</tr>
<tr>
<td>JGraph</td>
<td>18,373</td>
<td>Aspectizable interfaces</td>
</tr>
<tr>
<td>PetStore</td>
<td>17,032</td>
<td>Pointcut extraction</td>
</tr>
<tr>
<td>JS</td>
<td>13,979</td>
<td>Pointcut extraction</td>
</tr>
<tr>
<td>JAccounting</td>
<td>11,676</td>
<td>Pointcut extraction</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>514,624</strong></td>
<td></td>
</tr>
</tbody>
</table>
The second sub-package (java.awt) contains classes for creating graphic user interfaces and for painting graphics and images. The implementations of some layout managers and default event handlers are also included. Its sub-package java.awt.geom provides classes for defining and performing operations on two-dimensional objects. All these classes provide the general, basic behavior needed by advanced window-based environments (such as the swing environment), or by users who define their own one. Thus, similarly to the package java.util, these classes implement functionalities that are possibly required in a generic usage context.

One of the most important case studies is JHotDraw, because it was used in the assessment of many different techniques. JHotDraw is a framework offering two dimensional drawing facilities. It consists of approximately 18,000 non-commented lines of code and 2,000 methods. JHotDraw contains a quite well structured source code, because it has been written as an example to show good practice in using design patterns. So, it is a good candidate as a case study for aspect mining. In fact, most of the concerns are expected to be already separated. Those still tangled are probably associated to intrinsic limitations of OOP and aspects may be a good alternative for them.

A single crosscutting concern was manually identified in some of these applications. Based on the available documentation as well as code reading, we determined the concern that looked the best (most obvious and natural) candidate for aspectization. The concern code was annotated manually, without any aspect mining support. For JHotDraw and for PetStore, the refactored concern (respectively, Undo and Contract enforcement) has been used as a candidate aspect in other studies [48]. For JSpider and JAccounting, the selected concern is one of the typical examples of aspects reported in the literature on AOSD (respectively, Logging and Transaction management).
JHotDraw/Undo: In JHotDraw the execution of commands for the manipulation of graphical objects relies upon a hierarchy of classes which descend from AbstractCommand. Each class in the hierarchy implements (or inherits) methods to set/get the undo activity, i.e., to store or retrieve the information necessary to undo the command. Each drawing tool available in JHotDraw is responsible for setting/getting the undo activity upon command execution, the related code being highly scattered and tangled with the base code.

PetStore/Contract enforcement: PetStore uses XML as the standard representation format for its data. During XML parsing, the actions associated with the normal execution flow are mixed with checks on the occurrence of errors. If the document does not respect the prescribed format (both the syntactic constraints and the semantic contracts), an exception is raised, which requires corresponding exception management code. The contract enforcement and exception handling code is highly scattered, replicated and tangled with the base code.

JSpider/Logging: JSpider keeps track of the activities it performs and of the problems encountered into a log file. The statements to write logging information are highly scattered, since they occur wherever an activity takes places and an error can happen. The code fragments that implement logging are often replicated and they have little to do with the core functionalities of this application.

JAccounting/Transaction management: Transaction management is implemented in JAccounting using a basic idiom, which is replicated and slightly modified at several places in the program. The idiom consists of the following steps: (1) a session is started; (2) one or more operations are executed; (3) the operations are committed; (4) if an error occurred during (1), (2), (3) and an exception was raised, operations are un-done through rollback; (5) in case of both normal and exceptional termination,
the session is closed. This idiom is mixed with the core logic, which is correspondingly quite obscured.

### 5.2 Dynamic Aspect Mining

In order to apply the proposed technique to the target application (JHotDraw) a set of use-cases must be defined first. Since no use-case document about the case study is publicly available, the use-cases have been extracted from the application user manual. A use-case was defined for each main functionality. The aspect mining was then conducted on the traces coming from the execution of 27 use-cases, exercising a total of 1,262 methods. Thus, the formal context is composed of 27 objects and 1,262 attributes. Such a formal context produces a concept lattice containing 1,514 nodes. The visual inspection of such a lattice is not feasible, so a script was implemented to extract those nodes that verify the scattering and tangling candidates.

In total 11 concepts verify these properties, so they are called *use-case specific concepts*. They have been manually filtered in order to separate false positive from actual crosscutting concerns. Table 5.2 shows the result after filtering. Four crosscutting concerns have been classified as good aspect candidates. Among them, the most interesting one is the *Undo* concern.

In order to verify whether other crosscutting concerns are hidden in the

<table>
<thead>
<tr>
<th>Crosscutting Concern</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo</td>
<td>33</td>
</tr>
<tr>
<td>Bring to front</td>
<td>3</td>
</tr>
<tr>
<td>Send to back</td>
<td>3</td>
</tr>
<tr>
<td>Connect text</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 5.2: JHotDraw Results (use-case specific concepts).
code, we relaxed the conditions that a concept must verify to be considered a crosscutting concern. In this way, a larger set of aspect candidates is obtained. We dropped the first condition: a concept is no longer required to be a use-case specific concept. A node represents a crosscutting concern, and thus it is called a \textit{generic concept}, if it satisfies the remaining two conditions:

- The node is labeled by computational units (methods) that belong to more than one module (class).
- Different computational units (methods) from a same module (class) label also other concepts.

In this way we take into account also those concepts that are not directly related to the limited set of considered use-cases. Generic concepts can be regarded as associated with a hypothetic use-case coming from the (even quite complex) composition of the actually considered use-cases (through intersection/union). Manual inspection is required not only to filter out false positives, but also to group together similar concepts that are split in multiple nodes, although they can be mapped to the same concern. Moreover, a name is assigned to those generic concepts that do not correspond to any specific use-case. The identified set of 56 generic concepts has been filtered/grouped into the 18 aspect candidates shown in Table 5.3.

In order to manually filter the data coming from the tool, these guidelines have been followed:

- computational units must be associated with a single well-defined functionality that can be given a short description or name;
- some of the classes involved in the functionality have a primary responsibility different from this functionality (i.e., the functionality crosses the principal decomposition).
### Table 5.3: JHotDraw Results (generic concepts).

<table>
<thead>
<tr>
<th>Crosscutting Concern</th>
<th>Concepts</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>Bring to front</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Send to back</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Connect text</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Persistence</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>Manage handles</td>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td>Manage figure change event</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Move figure</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Command executability</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>Connect figures</td>
<td>1</td>
<td>55</td>
</tr>
<tr>
<td>Figure observer</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>Add text</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>Add URL to figure</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Manage figures outside drawing</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Get attribute</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Set attribute</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Manage view rectangle</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Visitor</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>
Even if the criteria have been stated and followed as much as possible, a level of subjectivity still remains, because deciding whether a functionality does or does not represent a crosscutting concern depends on how the design of the application is perceived by the user, thus different developers could provide different solutions.

Comparing the results for the first and the second experiment (Table 5.2 and Table 5.3) a lot (14) of novel aspect candidates appear. Some of them are very small (Manage figures outside drawing, Get/Set attribute and Manage view rectangle contain just 2 methods), whereas some of them are quite large (60 methods in Manage handles and 55 in Connect figures). Most of these new concerns are composed by only one formal concept from the lattice. Only four concerns are the composition of multiple concepts (Undo, Manage handles, Manage figure change event and Figure observer). Considering the common aspect candidates that have been found in the two experiments, three of them are exactly the same, while one is slightly different. In fact, in the second experiment the Undo concern becomes larger, because two formal concepts contribute to the seed method list.

The methods reported by the proposed aspect mining technique (counted in Table 5.3 third column) do not represent a full concern. They rather represent portions of it (dynamic analysis is partial). They have to be considered concern seeds, i.e., starting points to be refined during the actual separation of the concerns in the application.

The Undo concern seeds have been analyzed in detail in order to mea-
sure the accuracy of the proposed technique, and to understand how the identified seeds relate to the corresponding complete crosscutting concern. The case study application has been manually inspected and the full Undo concern has been marked and compared with the seeds (Table 5.4). The user was asked to specify all the methods belonging to this tangled functionality. The complete list contains 47 methods. The comparison of the seeds against the gold standard produced by the user gives a recall (proportion of methods marked by the user which are reported by the mining technique) of 49% and a precision (proportion of correct seeds among all those reported by the mining technique) of 64%. These values suggest that the quality of the Undo seed is quite promising, but human intervention is anyway required to correct and expand the automatically discovered seeds into the corresponding full concern.

5.3 Aspect mining comparison and combination

In this section, results of dynamic aspect mining on JHotDraw are compared with the results obtained using techniques proposed by other authors: fan-in analysis [48] and identifier analysis [66]. A description of these techniques can be found in Chapter 2. The limitations of each technique as well as their complementarity are discussed at the end of the section.

5.3.1 The fan-in analysis experiment

Fan-in analysis [48] is based on the assumption that a method called from many different places (i.e. with a high fan-in) represents a seed for a crosscutting concern, in that the call sites are spread throughout the system. Only methods whose fan-is is above a given threshold are reported as potential crosscutting concerns.

Fan-in analysis first performs a number of successive steps to filter the
methods in the analyzed system. The threshold-based filtering, which selects methods with high fan-in values, keeps around 7% of the total number of methods in JHotDraw. The filters for accessors and utility methods eliminates around half of the remaining methods. In the remaining subset, more than half of the methods (52%) are categorized as seeds, based on manual analysis.

Table 5.5 gives an overview of the types of crosscutting concerns that were identified. Several of these concern types, such as consistent behavior or contract enforcement [61], have more than one instance in JHotDraw; that is, multiple unrelated (crosscutting) concerns exist that conform to the same general description. For example, one instance of contract enforcement checks a priori conditions for a command’s execution, while another instance verifies common requirements for activating drawing tools. The number of different instances that were detected is indicated in the second column.

We distinguish three different ways in which the fan-in metric can be associated with the crosscutting structure of a concern implementation (also indicated in Table 5.5):

<table>
<thead>
<tr>
<th>Concern type</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistent behavior</td>
<td>4</td>
</tr>
<tr>
<td>Contract enforcement</td>
<td>4</td>
</tr>
<tr>
<td>Undo</td>
<td>1</td>
</tr>
<tr>
<td>Persistence and resurrection</td>
<td>1</td>
</tr>
<tr>
<td>Command design pattern</td>
<td>1</td>
</tr>
<tr>
<td>Observer design pattern</td>
<td>1</td>
</tr>
<tr>
<td>Composite design pattern</td>
<td>2</td>
</tr>
<tr>
<td>Decorator design pattern</td>
<td>1</td>
</tr>
<tr>
<td>Adapter design pattern</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.5: Summary of the results of the fan-in analysis experiment.
1. The crosscutting functionality is implemented through a method and the crosscutting behavior resides in the explicit calls to this method. Examples in this category include consistent behavior and contract enforcement.

2. The implementation of the crosscutting concern is scattered throughout the system, but makes use of a common functionality. The crosscutting resides in the call sites, and can be detected by looking at the similarities between the calling contexts and/or the callers. Examples of concerns in this category are persistence and undo [48].

3. The methods reported by the fan-in analysis are part of the roles superimposed to classes that participate in the implementation of a design pattern. Many of these roles have specific methods associated to them: the subject role in an Observer design pattern is responsible to notify and manage the observer objects, while the composite role defines specific methods for manipulating child components. In general, establishing a relation between these seed-methods and the complete concern they belong to might require a better familiarity of the human analyzer with the code being explored, than for the previous two categories. However, many of these patterns are well-known and have a clear defined structure, which eases their recognition [37].

5.3.2 The identifier analysis experiment

In [66] naming convention is used to identify crosscutting concerns. Identifiers are split into words according to the contained capital characters (camel casing). For instance, the class name QuotedCodeConstant generates the substrings quoted, code and constant. Formal Concept Analysis is then applied to this word list to group together source code entities, when they share similar names. Groups are then manually filtered to discard
false positive. Remaining groups correspond to the aspectual views that should guide developers in the crosscutting concern identification.

Applying the identifier analysis technique on JHotDraw yielded 230 concepts and took about 31 seconds when using a threshold of 4 for the minimum number of elements in a concept. With a threshold of 10, the number of concepts produced was significantly less: only 100 concepts remained after filtering, for a similar execution time.\(^1\) In both cases, 2193 objects and 507 attributes were considered. It is a good sign that the number of attributes is significantly smaller than the total number of objects considered,

\(^1\)Whereas the threshold of 4 was chosen arbitrarily, the threshold of 10 was determined experimentally: below that threshold the amount of concepts that were regarded as noise was significantly higher than above the threshold.
as it implies that there is quite some overlap in the identifiers of the different source-code entities, which was one of the premises of the identifier analysis technique.

The manual part of the experiment, i.e. deciding which concepts were real seeds, was much more time-consuming. Overall, this took about three days for the experiment with threshold 4, where 230 seed candidates needed to be investigated. For each of the discovered concepts, the code of the entities in its extent had to be inspected to decide whether (most of) these entities addressed a similar concern.

Table 5.6 presents some of the seeds discovered by manually analyzing the classes and methods belonging to the extent of the concepts produced by the FCA algorithm. The first column names the concern, the second column shows the identifiers shared by the objects belonging to the concept(s) corresponding to that concern. The third column shows the size of the extent for each concept. Finally, for illustration purposes, the fourth column shows some program entities appearing in the extent of the discovered concepts.

Out of 230 candidate seeds, 41 seeds were retained, when using a threshold of 4 for the minimum number of objects in a concept. These discovered concerns were classified in three different categories:

1. Some of these concerns looked like aspects in the more traditional sense (e.g., observer, undo and persistence).

2. Many other concerns seemed to represent a crosscutting functionality that was part of the business logic (e.g., drawing figures, moving figures). The distinction between these two first categories was somewhat subjective, however.

3. Three Java-specific concerns were discovered (e.g., iterating over collections) that are difficult to factor out into an aspect because they
Some concerns have been selected that were detected by all three techniques, as well as a representative set of concerns that were detected by some techniques but not by others. This allows to clearly pinpoint the strengths and weaknesses of each individual technique.

Table 5.7 summarises the concerns we selected. The first column names the concern. The other columns show by what technique(s) the concern was discovered: if a technique discovered the concern, we put a + sign in the corresponding column, otherwise a - sign is in the table.

**Observer**

The Observer design pattern is an example of a concern reported by all techniques. Other examples include *Command execution*, *Undo* functionality and *Persistence*, whose implementation in JHotDraw is described in [48]. Their identification should come as no surprise, because they corre-
spond to well-known aspects, frequently mentioned in the AOSD literature, or to functionalities for which an AOSD implementation looks quite natural.

Concerns identified by all three techniques are probably the best starting point for migrating a given application to AOSD, because developers can be quite confident that the concern is very likely to be an aspect. However, the fact that only four of such concerns were discovered, stresses the need for an approach that combines the strengths of different techniques.

**Contract enforcement / consistent behavior**

The *contract enforcement* and *consistent behavior* concerns [61] generally describe a common functionality required from, or imposed on, the participants in a given context, such as a specific pre-condition check on certain methods in a class hierarchy. An example from the JHotDraw case is the Command hierarchy for which the `execute` methods contain code to ensure the pre-condition that an ‘active view’ reference exists (is not null).

We classify these concerns as a combination of contract enforcement and consistent behavior since these types often have very similar implementations, and choosing a particular type depends mainly on the context and on (personal) interpretation.

Fan-in analysis is particularly suited to address this kind of scattered, crosscutting functionalities, which involve a large number of calls to the same method, while the other two techniques potentially miss it. In fact, contract enforcement and consistent behavior are usually associated with method calls that occur in *every* execution scenario, so that they cannot be discriminated by any specific use-case. On the other hand, identifier analysis will miss those cases where the methods that enforce a given contract or ensure consistent behavior do not share a common naming scheme.
CHAPTER 5. ASSESSMENT  

5.3. ASPECT MINING COMPARISON

Command execution

This concern deals with the executability and the actual execution of objects whose class belongs to the Command hierarchy. Identifier analysis identified a concept which contains exactly the execute methods in the Command hierarchy. Dynamic analysis identified the classes containing isExecutable methods. Indeed, the execute methods all have the same name and manual inspection showed they exhibit similar behavior: they nearly all make a super call to an execute method, invoke a checkDamage method and (though not always) invoke a setUndoActivity and getUndoActivity method. A similar argument can be made for isExecutable.

Hence, whereas identifier and dynamic analysis may not detect the more generic Contract enforcement / Consistent behavior aspect directly, they can identify some locations (pointcuts) where potentially such an aspect could be introduced.

Bring to front / Send to back

The functionality associated with this concern consists of the possibility to bring figures to the front or send them to the back of an image. When exercised, it executes specific methods that have a low fan-in, hence they were not detected by fan-in analysis. Identifier analysis also missed them, because there were not enough methods with a sufficiently similar name to surpass the threshold. Hence, dynamic analysis is the only technique that identified this concern. This example is a good representative of crosscutting concerns that are reported only by dynamic analysis: whenever the methods involved in a functionality are not characterized by a unifying naming scheme (or there are not enough of them), neither do they have high fan-in, the other two techniques are likely to fail.
Manage handles

A crosscutting functionality is responsible for managing the handles associated with the graphical elements. Such handles support interactive operations, such as resizing of an element, conducted by clicking on the handle and dragging the mouse. This seed is interesting because it is detected by dynamic analysis and by identifier analysis, but in different ways. Identifier analysis detects this concern based on the presence of the word “handle” in identifiers. Consequently, it misses methods such as `north()`, `south()`, `east()`, `west()`, which are clearly related to this concern, but do not share the lexicon with the others. On the other hand, dynamic analysis reports both the latter methods and (some of) those containing the word “handle”. However, since not all possible handle interactions have been exercised, the output of dynamic analysis is partial and does not include all the methods reported by identifier analysis.

The `manage handles` concern was missed by the fan-in analysis because the calls are too specific: they are similar but different calls instead of one single called method with a high fan-in.

Moving figures

The three techniques discard concerns on different bases: some of the concerns are filtered automatically while others are excluded manually. The `move figures` concern, seeded by the `moveBy` method in the `Figure` classes, is one example where different, subjective decisions can be made depending on whether the concept is classified either as a candidate aspect or as part of the principal decomposition. The `moveBy` methods allow to move a figure with a given offset. When applying fan-in analysis it was argued that the original design seems to consider this functionality as part of a `Figure`’s core logic. When applying the other two methods it was consid-
erred as part of a crosscutting functionality and it was included in the list of reported seeds.

5.3.4 Limitations

As a consequence of applying each technique to the same case, some of the limitations of the respective techniques have become obvious. For example, we obtained a better idea of potential “false negatives”, i.e. concerns that were not identified by a particular technique but that were identified by another. Below, we summarise some of the discovered limitations. In the next subsection we then describe how to partly overcome these limitations by combining different techniques.

Fan-in analysis

Fan-in analysis mainly addresses crosscutting concerns that are largely scattered and that have a significant impact on the modularity of the system. The downside of this characteristic is that concerns with a small code footprint and thus with low fan-in values associated, will be missed. For example, the identification of Observer design pattern instances is dependent on the number of classes implementing the observer role. These classes contain calls to specific methods in the subject class for registering as listeners to the subject’s changes. The number of observer classes will determine to a large extent the number of calls to the registration method in the subject role. A collateral effect is the anticipated unsuitability of the technique for analysing small case studies.

Identifier analysis

Identifier analysis tends to produce a lot of detailed results. However, these results typically contain too much noise (false positives), so a more effec-
5.3. ASPECT MINING COMPARISON CHAPTER 5. ASSESSMENT

tive filtering of the discovered concepts, as well as of the elements inside those concepts, is needed. In addition, the discovered concepts are often incomplete, in the sense that they do not completely “cover” an aspect or crosscutting concern. Often, more than one concept is needed to describe a single concern, as was the case of the Observer aspect. The individual concepts themselves may also need to be completed with additional elements that are not contained in those concepts. This was the case for the Undo aspect: in addition to the methods with ‘undo’ or ‘undoable’ in their name, some of the methods calling these undo methods need to be considered as part of the core aspect as well.

Dynamic analysis

Dynamic analysis is partial (i.e., not all methods involved in an aspect are retrieved), being based on specific executions, and it can determine only aspects that can be discriminated by different execution scenarios (e.g., aspects that are exercised in every program execution cannot be detected). Additionally, it does not deal with code that cannot be executed (e.g., code that is part of a larger framework, but that is not used in a specific application).

5.3.5 Complementarity

The three techniques address symptoms of crosscutting functionality, such as scattering and tangling, in quite different ways. As shown in Table 5.8, fan-in analysis and dynamic analysis show largely complementary result sets: among the 30 concerns identified by either dynamic or fan-in analysis, only 4 are identified by both techniques. This is an expected result. Fan-in analysis focuses on identifying those methods that are called at multiple places. However, when a method is called many times, it is likely to occur in most (if not all) execution traces. Hence, no specific use-case can be
CHAPTER 5. ASSESSMENT  5.3. ASPECT MINING COMPARISON

<table>
<thead>
<tr>
<th>Technique</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic analysis</td>
<td>18</td>
</tr>
<tr>
<td>Fan-in analysis</td>
<td>16</td>
</tr>
<tr>
<td>Dynamic analysis $\cup$ Fan-in analysis</td>
<td>30</td>
</tr>
<tr>
<td>Dynamic analysis $\cap$ Fan-in analysis</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.8: Concerns identified by either dynamic or fan-in analysis.

defined to isolate the associated functionality, and dynamic analysis will fail to identify it as a seed.

Identifier analysis is the least discriminating of the three techniques and has a large overlap with the other two techniques. When a concern can be identified through fan-in analysis and/or dynamic analysis, identifier analysis can often isolate it too, since a common lexicon is often used in the names of the involved methods.

These observations can be used to propose new aspect mining techniques as clever combinations of the three individual techniques.

Fan-in analysis and dynamic analysis are largely complementary, and address different symptoms of crosscutting. An obvious and interesting combination of these techniques thus consists of simply applying each technique individually and taking the union of the results. Additionally, the seeds in the intersection of the results (if any) are likely to represent the best aspect candidates, because both techniques identify them. This was illustrated in our experiment, in which both techniques identified the Observer, Undo, Persistence and Command execution candidates.

With regards to combinations involving identifier analysis, two interesting observations can be made. First, the manual intervention required by identifier analysis is very time-consuming and is not justified by the fact that it produces more interesting results. This makes the technique less suited than the others for large(r) cases. Second, both fan-in analysis
and dynamic analysis identify only candidate seeds that serve as a starting point for seed expansion. Dynamic analysis in particular suffers from this problem as it is based on a (necessarily partial) list of execution scenarios. Similarly, fan-in analysis is only focused on invocations of high fan-in methods, which represent just a portion of the whole concern. Interestingly, while performing fan-in analysis and dynamic analysis, we observed that the classes and methods in the seed expansion often exhibited similar identifiers.

Consequently, we believe better results can be obtained if we use identifier analysis as a seed expansion technique for the seeds identified by either fan-in analysis or dynamic analysis, or by the seeds identified by both these techniques. In this way, the search space for identifier analysis is reduced significantly, and more automation is provided for the manual seed expansion needed by both fan-in analysis and dynamic analysis. A final manual refinement step is anyway necessary, since the expanded seeds may contain false positives and negatives.

In the next sub-section, we describe how to combine fan-in analysis with identifier analysis, dynamic analysis with identifier analysis, and the union of fan-in analysis and dynamic analysis with identifier analysis.

5.3.6 Definition of the combined techniques

The combined techniques work as follows:

1. Identify interesting candidate seeds by applying fan-in analysis, dynamic analysis or both to the application;
   - For candidate seeds identified by dynamic analysis, (manually) filter out those methods that do not pertain to the concern;

2. For each method in the candidate seed, find its enclosing class, and
compute the identifiers occurring in the method and the class name, according to the algorithm used by identifier analysis;

3. Apply identifier analysis to the application, and search for the concept, among the concepts it reports, that is “nearest”. The nearest concept is the concept that contains most of the identifiers generated in the previous step. If more than one nearest concept exists, take the union of all their elements.

4. Add the methods contained in the nearest concept(s) to the candidate seed (seed completion).

5. Revise the expanded list of candidate seeds manually to remove false positives and add missing seeds (false negatives).

In what follows, we experimentally validate these techniques on the JHotDraw case.

5.3.7 Analysis indicators

Before applying the combined techniques, we define two measures to validate the results. The goal is to measure how identified seeds change in terms of precision and recall. Unfortunately, this requires information about all crosscutting concerns present in the application, and this is not always available. Therefore, we have chosen alternative metrics, which we call recalled methods and precision.

**Recalled methods** is the number of methods reported in a seed that actually belong to the crosscutting concern.

**Precision** is the percentage of a seed’s recalled methods with respect to the total number of methods in the seed. This indicator estimates how difficult it is to spot a concern in the methods provided by the seed.
<table>
<thead>
<tr>
<th>Concerns</th>
<th>Undo</th>
<th>Command execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique</td>
<td>Recalled Methods</td>
<td>Precision</td>
</tr>
<tr>
<td>Dynamic analysis</td>
<td>23</td>
<td>64%</td>
</tr>
<tr>
<td>Fan-in analysis</td>
<td>3</td>
<td>(100%)</td>
</tr>
<tr>
<td>Dyn ∪ Fan-in</td>
<td>24</td>
<td>63%</td>
</tr>
<tr>
<td>Dyn + Identifier</td>
<td>183</td>
<td>55%</td>
</tr>
<tr>
<td>Fan-in + Identifier</td>
<td>94</td>
<td>100%</td>
</tr>
<tr>
<td>(Dyn ∪ Fan-in) + Identifier</td>
<td>183</td>
<td>55%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Persistence</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique</td>
<td>Recalled Methods</td>
<td>Precision</td>
</tr>
<tr>
<td>Dynamic analysis</td>
<td>29</td>
<td>97%</td>
</tr>
<tr>
<td>Fan-in analysis</td>
<td>6</td>
<td>(100%)</td>
</tr>
<tr>
<td>Dyn ∪ Fan-in</td>
<td>32</td>
<td>97%</td>
</tr>
<tr>
<td>Dyn + Identifier</td>
<td>104</td>
<td>100%</td>
</tr>
<tr>
<td>Fan-in + Identifier</td>
<td>104</td>
<td>100%</td>
</tr>
<tr>
<td>(Dyn ∪ Fan-in) + Identifier</td>
<td>104</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 5.9: Recalled methods and precision before and after combination.

With respect to the definitions above, it is important to remark that for fan-in, two interpretations of seeds are possible: the first takes only the callees with high fan-in into account; the second interpretation includes, besides the callees with high fan-in, also all callers to these methods. Data reported in this study refer to the former interpretation.

### 5.3.8 Experimental results

Table 5.9 shows the values of the indicators before and after the completion experiment. Fan-in reports always a 100% precision, because this method includes a manual filtering of the method list automatically provided by this approach, so no false positives can be present. For this reason the
values have been enclosed within brackets. Although the completion technique can be applied to all concerns identified by either fan-in analysis or dynamic analysis, we performed the experiment only on the concerns identified by all three techniques. The sole reason is that we need to assess how the completion technique influences the recalled methods and precision indicators as compared to their initial values, which can only be done for the *Undo*, *Command execution*, *Persistence* and *Observer* concerns.

A deeper look into the results of the completion with identifier analysis reveals interesting information: For the *Undo* concern, the results of both fan-in analysis and dynamic analysis improve a lot in terms of recalled methods (from 23 and 3 up to 183 and 94). There is a negative impact on the precision for (completed) dynamic analysis (from 64% down to 55%), but the precision for fan-in plus identifier analysis remains at 100%. For the *Command execution* and *Persistence* concerns, the number of recalled methods increases significantly for the completion technique (from 20 and 3 up to 132 and from 29 and 6 up to 104), while the precision remains at the same level.

For the *Observer* concern, the results are less encouraging than for the other concerns. Even though the number of recalled methods increases for the completion technique, the precision drops to an unacceptable level (from 100% down to 14% and 15%). Clearly, completion does not provide a good expansion of the original seeds. Closer inspection reveals that no clearly distinctive naming convention has been used to implement the *Observer* concern. The *Undo*, *Command execution* and *Persistence* concerns employ distinctive identifiers such as `undo/undoable`, `execute/command` and `store/storable`, which are used extensively only within the concern implementation. Consequently, the completion provided by identifier analysis gives good seed expansions. However, the identifiers used for the *Observer* concern are the more general `figure/update/...` that are used
extensively throughout the application, and not only in the concern implementation. Therefore, identifier analysis is not able to provide a good expansion for the seeds found by the other techniques.

### 5.4 Pointcut extraction

This section describes the results obtained by applying AOP-Migrator to four case studies. The migration task executed on them was designed to simulate a typical scenario of smooth migration toward AOSD. This consists of small incremental steps and is initially focused on a single, well known and clearly recognizable concern to be migrated. The aim is not to restructure the application completely according to the AOSD paradigm. Rather, a controlled and well identified modification is made, in order to assess its execution and the related opportunities for automation. The refactored code for the four case studies is available\(^2\).

Four applications (JHotDraw, PetStore, JSpider and JAccounting) have been used in this assessment. Their code has been inspected and the cross-cutting concerns described in Section 5.1 have been manually identified (JHotDraw/Undo, PetStore/Contract enforcement, JSpider/Logging and JAccounting/Transaction management).

#### 5.4.1 Refactoring

Table 5.10 shows some statistics on the OO transformations that have been used to enable the application of the six proposed object-to-aspect refactorings. Overall, they represent a remarkable proportion of the total refactoring effort (40% of the total refactorings are enabling OO-transformations). Among them, around 1/3 of the cases are either *Statement Reordering* or *Extract Method*. Thus, the experimental results confirm our intuition that

\(^2\)The case studies can be found on the tool Web site [http://se.itc.it/aop-migrator](http://se.itc.it/aop-migrator)
these two refactorings play a major role in the migration of real code. The remaining cases are almost equally split between ad-hoc transformations, that depend on the specific organization of the code and must be carried out manually, and other OO transformations, that are either already available in Eclipse as OO transformations (e.g., Extract Local Variable, Promote Local Variable to Class Field) or that could be easily implemented and added to the existing ones (e.g., Push Assignment). It is interesting to note that in total 2/3 of the enabling OO-transformations are automated (or automateable). This means that although the enabling OO-transformations represent an important cost in the migration process, there is space to support their execution through automated transformations.

Among the enabling refactorings, Extract Method is the one that may be potentially detrimental to the code quality. In practice, what we observed in our four case studies is that the extracted methods typically represent meaningful, functionally cohesive, abstractions. Often they belong to long methods (hence the difficulty of finding unique join points), that can be better organized into smaller units.

Table 5.11 shows the distribution of the six proposed refactorings from objects to aspects, when applied to the four case studies. Three of them
### 5.4. POINTCUT EXTRACTION

#### CHAPTER 5. ASSESSMENT

<table>
<thead>
<tr>
<th>Case study</th>
<th>Begin/end</th>
<th>Before/after call</th>
<th>Conditional</th>
<th>Pre Return</th>
<th>Wrapper</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>26</td>
<td>41</td>
<td>4</td>
<td>3</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>PetStore</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>76</td>
</tr>
<tr>
<td>JSpider</td>
<td>80</td>
<td>67</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>76</td>
</tr>
<tr>
<td>JAccounting</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td><strong>Overall aspect refactoring</strong></td>
<td><strong>136</strong></td>
<td><strong>108</strong></td>
<td><strong>23</strong></td>
<td><strong>8</strong></td>
<td><strong>25</strong></td>
<td><strong>91</strong></td>
</tr>
<tr>
<td><strong>Aspect over total refactorings</strong></td>
<td><strong>35%</strong></td>
<td><strong>28%</strong></td>
<td><strong>6%</strong></td>
<td><strong>2%</strong></td>
<td><strong>6%</strong></td>
<td><strong>23%</strong></td>
</tr>
</tbody>
</table>

Table 5.11: Aspect refactorings applied to the four case studies.

appear to play a major role: Extract Begin/end, Extract Before/after Call and Extract Exception. Among the remaining three refactorings, Extract Conditional and Extract Wrapper gave some contribution to migration, accounting in total for around 12% of the refactoring instances. Pre Return is the least used refactoring.

The wrappers refactored in JHotDraw serve typically the purpose of adding the undo functionality to a command. For example, when the Edit menu is created, the menu items added to it are commands (e.g., PasteCommand). Instead of adding such commands directly, an instance of UndoableCommand is added, which wraps the actual command. Both PasteCommand and UndoableCommand implement the interface Command.

In JAccounting, the typical transaction management idiom (see description above) involves intercepting the beginning/end of methods to initiate/terminate a session. Moreover, exception handling associated with transaction failures is moved to the aspect. Hence the high number of Begin/end and Exception reported in Table 5.11.

The high number of refactorings of type Conditional applied to PetStore is due to the way contract enforcement was implemented. During the traversal of XML trees, constraints are checked in a conditional statement.
where nodes and attributes are retrieved and verified for sanity. If the check fails, an exception is raised to signal the error. After refactoring, such checks are completely performed within an aspect.

Overall, the data in Tables 5.10 and 5.11 suggest that in practical cases a limited number of automated refactorings from objects to aspects is sufficient to migrate an existing application. At the same time, they indicate that the application cannot be migrated without transformation. In order to untangle the base code from the concern to be aspectized and in order to let the aspect intercept proper join points in the base code, some enabling OO-transformations are necessary. However, the majority of them can be automated and added (when not already available) to the list of those supported by existing tools (such as Eclipse).

### 5.4.2 Pointcut Abstraction

The second step of the extraction is the merge of the individual advice instances into the final aspect. In general, pointcut abstraction is a research area in itself, which deserves further study and investigation [31, 65]. However, given the structure of the pointcut instances extracted by AOP-Migrator, the simple syntactic merge strategy described in the previous chapter is often sufficient. In order to make a preliminary assessment of its potential to improve the output of the refactoring, we consider merging the aspect instances extracted from JAccounting.

The JAccounting code contains 45 instances (i.e., distinct code fragments) of the transaction aspect that capture creating a new transaction, committing a transaction, and aborting a transaction. After pointcut abstraction, these are intercepted using four pointcuts, as aborting a transaction sometimes generates a stack dump.

Figure 5.1 illustrates the result of merging the 15 create transaction pointcut and advice instances. All 15 instances of the pointcut include Line
5.4. POINTCUT EXTRACTION

CHAPTER 5. ASSESSMENT

```java
public aspect Transaction percflow(p_createTransaction, ...) {
    net.sf.hibernate.Transaction tx;

    pointcut p_createTransaction() {
        call(Session SessionFactory.openSession())
        && (execution(String ProductsPage.perform2())
            || execution(String InvoicePage.perform2())
            || execution(String RecurrencePage.perform2())
            || execution(String PaymentPage.perform2())
            || execution(String CustomerDetails.perform2())
            || execution(String CustomerForm.perform2())
            || execution(Account DefaultAccountSetup.
                createInternalAccount())
    };

    after()
    returning(Session sess)
    throws net.sf.hibernate.HibernateException : p_createTransaction()
    { tx = sess.beginTransaction(); }
}
```

Figure 5.1: Example of merged advices from JAccounting.

6 “call(Session SessionFactory.openSession())”; thus, one copy appears in the merged pointcut. The 15 transaction instances are created in 7 different methods whose context is captured by Lines 7 – 14. Together with the merged code for committing and aborting a transaction, the resulting aspect succinctly captures the “transaction concept”.

5.4.3 Separation of Concerns

To gain a better appreciation for the impact of aspect refactoring, this section presents data from two empirical investigations. The first considers dependence metrics from all four of the case study programs, while the second looks specifically at JHotDraw.

Definition 1: A pair \((X, Y)\) represents a Contains Dependence if \(X\) is a class and \(Y\) is the type of one of \(X\)’s data members.
5.4. POINTCUT EXTRACTION

Table 5.12: Entities (types) and relationships (dependences) in the base code.

<table>
<thead>
<tr>
<th>Case study</th>
<th>base types</th>
<th>concern types</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>289</td>
<td>37</td>
</tr>
<tr>
<td>PetStore</td>
<td>285</td>
<td>0</td>
</tr>
<tr>
<td>JSpider</td>
<td>241</td>
<td>10</td>
</tr>
<tr>
<td>JAccounting</td>
<td>87</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case study</th>
<th>base-to-base deps</th>
<th>base-to-concern deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>765</td>
<td>59</td>
</tr>
<tr>
<td>PetStore</td>
<td>585</td>
<td>0</td>
</tr>
<tr>
<td>JSpider</td>
<td>746</td>
<td>34</td>
</tr>
<tr>
<td>JAccounting</td>
<td>109</td>
<td>35</td>
</tr>
</tbody>
</table>

**Definition 2:** A pair \((X, Y)\) represents a *Uses Dependence* (or simply a *Dependence*) if \(X\) and \(Y\) are classes or interfaces and \(Y\) is mentioned explicitly in the body of \(X\), i.e., in declarations, allocation or cast statements within \(X\).

Clearly, Contains Dependences are also (Uses) Dependences, so the former are just a specific case of the latter. Dependences are computed as sets of pairs. This means that when they are counted, the number of instances of the same pair does not matter.

The dependence metrics defined above can be justified intuitively as an indicator of the comprehension effort required to understand the base code. In fact, a lower number of classes/interfaces and a lower number of relationships among them is expected to simplify the comprehension tasks. With regards to the understandability of the concern code, it should be noticed that after refactoring it is encapsulated into a single modular unit (an aspect). Correspondingly, understanding and modifying the concern code is expected to be simplified.

Table 5.12 shows the number of types (classes and interfaces) and the number of dependences, divided into base (base-to-base) vs. concern (base-to-concern). The dependence metrics are directional (see Definition 1 and
2 above) and we are interested in those outgoing from the base code (base-to-base and base-to-concern), since they are chosen to characterize the difficulty of comprehension of the base code. The effect of refactoring is the removal, among such dependencies, of those directed toward concern classes. In fact, the base code remains oblivious of the concern code after refactoring.

The data in Table 5.12 indicate a substantial reduction of the number of types and of the number of dependencies thanks to refactoring. The only exception is PetStore, for which the benefits of aspectization can be appreciated only at a lower granularity level. In fact, the internal logics of the methods involved in the refactorings is simplified once contract enforcing is moved to a separate aspect, although no change can be observed in terms of number of types or inter-type relationships.

We are aware of the fact that the chosen indicators of the benefits of refactoring are quite arbitrary and partial. Other metrics could be taken into account (e.g., complexity metrics of the refactored methods would account for the effects produced on PetStore). Moreover, evaluating the overall effects of refactoring from the point of view of the final user would require the execution of controlled empirical studies with human subjects. However, this preliminary quantitative investigation let us hypothesize benefits associated with a simplification of the base code and with the modularization of the refactored concern.

The second empirical investigation shows the impact of refactoring on a selected part of the class diagram for JHotDraw. Figure 5.2 shows the twelve classes and four interfaces involved in “undoing” figure transfer commands. The diagram is dominated by two inheritance hierarchies: one rooted at AbstractCommand and the other at UndoableAdapter. In addition, it shows the Contains and Uses dependences between classes.

The effect of refactoring the undo functionality to an aspect is shown in
Figure 5.2: Excerpt from JHotDraw’s class diagram.

Figure 5.2. The refactoring removes the gray portions of diagram, which includes removing 13 of 21 dependences, 6 of 12 classes, and 1 of 4 interfaces. Most of the classes deal directly with the undo activity and their removal is not unexpected, but there are also two classes, StandardFigureSelection and FigureEnumerator, that are only used in support of undo functionality. The removal of all gray elements from JHotDraw’s class diagram illustrates the simplification of the primary functionality (base classes) achieved using aspect refactoring.
5.4.4 Performances

After refactoring, the four considered programs have been compiled (using the AspectJ compiler provided by Eclipse) and run to check that they actually executed as before. The impact on the performances of these applications due to the migration to AOP was assessed. Three of them can be run only interactively and during their execution the user was unable to notice any performance degradation. Only one case study (JSpider) supports batch execution mode. It has been used to download several Web sites in batch mode. Each download was repeated using the original and the AOP version of the code. The observed execution times showed no specific tendency. Sometimes the AOP version was faster, sometimes the original version was so, sometimes no difference was observed. Our conclusion is that execution times for this application are dominated by the network latency and by the load of the involved Web servers. The local execution time is negligible, since JSpider spends most of its time receiving data from remote servers over the Internet.

Overall, in our four case studies, no performance degradation was observed for the refactored AOP code. However, this cannot be generalized to arbitrary applications. We are quite confident that applications whose execution time is dominated by user interaction or network communication are likely to show no perceivable performance difference once migrated to AOP.

5.4.5 Size

Table 5.13 shows the size changes associated with the refactoring executed on the four case studies. We measured the Lines of Code (LoC) of the classes involved in the refactoring. The size reduction is quite limited and has no impact on the overall size figures (see Table 5.1) for the base code.
<table>
<thead>
<tr>
<th>Case study</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>4661</td>
<td>4123</td>
</tr>
<tr>
<td>PetStore</td>
<td>1913</td>
<td>1823</td>
</tr>
<tr>
<td>JSpider</td>
<td>4945</td>
<td>4485</td>
</tr>
<tr>
<td>JAccounting</td>
<td>1680</td>
<td>1422</td>
</tr>
</tbody>
</table>

Table 5.13: Base code size (LoC) before and after refactoring, for the classes subjected to refactoring.

Moreover, it should be noticed that the size decrease reported in Table 5.13 gives raise to a corresponding increase of the aspect code size, which can be quantified around the same amount. Thus, the net size effect on the overall systems is null, while it is almost negligible if measured for the base code only. If considered in conjunction with the dependence analysis described above, the data in Tables 5.13 and 5.12 indicate that the base code size is only marginally reduced, but that the base code structure is substantially simplified.

5.4.6 Lessons Learned

The most critical phase in the migration from objects to aspects consists of enabling OO-transformations. These reformulate the base code, so that the selected concern can be more easily separated from the base code. They allow proper join points to be made available in the base code, letting the aspect reintroduce the concern in appropriate places. Untangling the concern from the base functionality might be a hard task (a similar result was reported by Marius Marin [47]), requiring deep understanding of the functionalities and code organization. Most of the ad-hoc, manual OO transformations fall into this category.

Once a given code fragment is “ready” for migration (i.e., after having applied the necessary enabling OO-transformations to it), the next steps
are quite easy and AOP-Migrator provides considerable guidance to the user when performing them. If only one of the six available refactorings is applicable, the transformation is usually straightforward. If there is more than one possibility, the decision normally can be made quite quickly (with the aid of the automatically computed priorities and previews). In the current migration process, the less supported part is the application of the enabling OO-transformations. If, on one hand, the end-point of the OO transformation is usually known (i.e., is one of the six object-to-aspect refactorings) then how to get there often involves hard decision making. Present systems provide little guidance for undertaking such an activity; there is room for further investigation and automated support.

From the point of view of AOSD, it is interesting to note that only a subset of all the features provided by aspects is actually used in the migrated application. In part, this is due to the lack of a sophisticated post-processing which applies aspect-to-aspect transformations in order to improve the quality of the generated code. However, we think that in general a small number of aspect oriented constructs is sufficient to migrate meaningful crosscutting concerns present in real applications. Once the execution of the base code can be intercepted at appropriate places (begin/end of method, before/after call, during exception handling, etc.), there is enough power to separate and modularize the crosscutting concerns implemented in real applications.

The migration study was somewhat constrained by the AspectJ implementation of the AOSD principles. The limited context exposure facilities of AspectJ forced the use of additional OO-transformations to “prepare” the code by means of enabling OO-transformations, such as Extract Method. These enabling transformations were also used to introduce aspect fields serving as copies of local variables. These were eventually removed from the base code and moved to the aspect. The limited ability of As-
pectJ to intercept the execution of the base code also necessitated some enabling transformations. However, we think that the alternative of offering more context exposure/join points could potentially give raise to more fragile aspect definitions, thus resulting in a lower quality of the aspect code. Each time we had to operate some transformation to make the base code interceptable at the right point, we found that it was possible to apply a very meaningful code restructuring that would have been advisable independently of the migration to aspects. Overall, the compromise between limited code intrusion and aspect power offered by AspectJ seems a reasonable one.

Evaluating the benefits produced by the separation of selected concerns into distinct aspects is a hard task and goes beyond the scope of the present experiment. However, the preliminary results that we have gathered show that a substantial simplification of the base code is achieved through refactoring. Class diagrams become much more meaningful and understandable, since the classes and inter-class relationships pertaining to the separated concern are removed. Moreover, a substantial number of types and inter-type dependences disappears from the base code, which becomes oblivious of the extracted concerns. Qualitatively, the code looks much more focused on its core functionality, which is supposed to affect positively its understandability. No substantial effect on size was observed and in our case studies no detrimental effect on the performances of the applications was experienced.

5.5 Aspectizable interfaces

5.5.1 Aspectizable interfaces identification

The possibility to identify and migrate interface implementations, that can be regarded as crosscutting concerns, has been tested on the java source
code of three packages from the standard library of the Java 2 Runtime Environment, Standard Edition (build 1.4.0-b92). As shown in Table 5.14, the three analyzed packages contain 131 classes, for a total of 44 199 Line Of Code (LOC), comments excluded. The 52 interfaces implemented by these classes (179 implementations in total) have been subjected to aspect mining. Then, those representing actual crosscutting concerns have been migrated to aspects.

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
<th>LOC</th>
<th>Int. impl.</th>
<th>Unique interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util</td>
<td>41</td>
<td>13,993</td>
<td>61</td>
<td>13</td>
</tr>
<tr>
<td>java.awt</td>
<td>76</td>
<td>24,425</td>
<td>97</td>
<td>35</td>
</tr>
<tr>
<td>java.awt.geom</td>
<td>14</td>
<td>5,781</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>131</strong></td>
<td><strong>44,199</strong></td>
<td><strong>179</strong></td>
<td><strong>52</strong></td>
</tr>
</tbody>
</table>

Table 5.14: Data set under analysis.

In order to assess the results obtained with the methods for the mining of aspectizable interfaces described in Chapter 3 (Sub-section 3.2.2) a solution of trusted accuracy is necessary. An expert was involved in defining such a
solution. He examined the class diagram, the source code and the Javadoc documentation in order to understand the role of each interface in each implementing class.

The output of the expert’s analysis consists, for each package, of its principal decomposition, in which classes are grouped together with the implemented interfaces, when these are devoted to its main functionalities. Such interfaces are mandatory in defining the classes’ predominant responsibilities. Interface implementations outside the principal decomposition are regarded as the expert’s aspects.

Each of the four aspect mining methods has been applied to all the classes under analysis, giving a response about which interface implementations to change into an aspect. All the solutions have been compared with the (complement of the) expert’s decomposition. They have been ranked in terms of precision and recall, where the former measures the proportion of correctly identified aspects over all the candidates produced by a given method, while the latter measures the proportion of correctly identified aspects over all those to be retrieved (i.e., those specified by the expert).

Table 5.15 shows the mean value of precision and recall computed for the classes contained in each package under analysis. The average values at the bottom are weighted by the number of classes in each package, since the considered packages belong to the same system, instead of being separate applications. In this way, the impact of small packages (such as java.awt.geom) on the final result is reduced.

The three methods External Package, String Matching and Unpluggability are fundamentally equivalent, having very similar values of precision and recall. The last one has the highest precision, but a slightly lower recall. The remaining method, Clustering, has a similar performance in terms of recall, but a much lower precision, making it substantially worse.
5.5. ASPECTIZABLE INTERFACES

### Table 5.15: Precision and recall of the four aspect mining methods.

<table>
<thead>
<tr>
<th></th>
<th>External Package</th>
<th>String Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precision</td>
<td>recall</td>
</tr>
<tr>
<td>java.util</td>
<td>88%</td>
<td>100%</td>
</tr>
<tr>
<td>java.awt</td>
<td>91%</td>
<td>91%</td>
</tr>
<tr>
<td>java.awt.geom</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>86%</strong></td>
<td><strong>95%</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Clustering</th>
<th>Unpluggability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precision</td>
<td>recall</td>
</tr>
<tr>
<td>java.util</td>
<td>73%</td>
<td>95%</td>
</tr>
<tr>
<td>java.awt</td>
<td>82%</td>
<td>89%</td>
</tr>
<tr>
<td>java.awt.geom</td>
<td>64%</td>
<td>93%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>76%</strong></td>
<td><strong>94%</strong></td>
</tr>
</tbody>
</table>

than the others.

A further investigation of unpluggability was obtained by running the modules *UnpluggableFields* and *UnpluggableInnerClasses*. Fields used only by interface methods have been detected by the module *UnpluggableFields* in 4 classes, for a total of 16 unpluggable fields. All of them are related to the implementation of the serialization concern (*Serializable* interface). Looking at the source code, we discovered that the purpose of these unpluggable data structures is to save specific state information, useful in a situation where complicated objects are restored from the serialized data.

The module *UnpluggableInnerClasses* detected 16 unpluggable inner classes. All of them are related to the accessibility concern (*Accessible* interface). In all these cases, we found the same pattern: an inner class specializes the abstract class *javax.accessibility.AccessibleContext*, used as return type by the accessibility interface method.

No field or inner class that the expert considered as a part of an interface implementation to be migrated was missed by our unpluggability modules (i.e., recall, as well as precision, is 100% for fields and inner classes to be
Experimental results indicate that the problem of identifying which interface implementations are suited for migration to aspects can be approached with simple methods (such as string matching), which are able to produce a very good starting point.

5.5.2 Refactoring the aspectizable interfaces

Refactoring of the aspectizable interfaces was applied to a larger proportion of the standard Java library (all packages below \texttt{java} in the package hierarchy of JDK) and to three open source programs, JHotDraw, FreeTTS and JGraph. Although necessarily limited in number, these examples range from component oriented classes to full applications, with intermediate cases, such as JHotDraw and JGraph, where a development framework is provided together with complete applications.

Among the aspect mining methods for the identification of the aspectizable interfaces that have been investigated (see previous Section), we decided to use the simplest one, based on regular expressions. We looked for the interface names matching the pattern "[A-Z][a-z]*ble", based on the observation that crosscutting functionalities are often expressed as "able"-ties (e.g., \textit{Accessible}, \textit{Serializable}). Such simple aspect mining technique allowed us to identify 8 out of the 9 aspectizable interfaces in the code under analysis (see Table 5.17), with a recall (fraction of aspects actually mined among those to be identified) equal to 89\%. Precision (fraction of correct aspects among those retrieved) was 53\%. Since the classification between interfaces in the principal decomposition and aspectizable interfaces was always very sharp, with almost no controversial case, we are quite confident that the degree of subjectivity in the output of this activity is low.

The fourth column of Table 5.16 gives the number of actually imple-
### Table 5.16: Features of the source code under analysis. The last three columns show the unique implemented interfaces, among them the aspectizable interfaces, and the number of implementations of the aspectizable interfaces.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>java.applet</td>
<td>833</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>java.awt</td>
<td>140,274</td>
<td>259</td>
<td>56</td>
<td>5</td>
<td>86</td>
</tr>
<tr>
<td>java.beans</td>
<td>14,560</td>
<td>78</td>
<td>15</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>java.io</td>
<td>24,194</td>
<td>67</td>
<td>6</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>java.lang</td>
<td>34,330</td>
<td>100</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>java.math</td>
<td>5,794</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>java.net</td>
<td>20,713</td>
<td>52</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>java.nio</td>
<td>11,645</td>
<td>34</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>java.rmi</td>
<td>8,601</td>
<td>49</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>java.security</td>
<td>27,475</td>
<td>111</td>
<td>12</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>java.sql</td>
<td>13,741</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>java.text</td>
<td>25,480</td>
<td>43</td>
<td>6</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>java.util</td>
<td>54,893</td>
<td>103</td>
<td>12</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>382,533</strong></td>
<td><strong>914</strong></td>
<td><strong>132</strong></td>
<td><strong>23</strong></td>
<td><strong>184</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>39,214</td>
<td>249</td>
<td>48</td>
<td>5</td>
<td>54</td>
</tr>
<tr>
<td>FreeTTS</td>
<td>31,099</td>
<td>113</td>
<td>22</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>JGraph</td>
<td>18,373</td>
<td>29</td>
<td>20</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>88,686</strong></td>
<td><strong>391</strong></td>
<td><strong>90</strong></td>
<td><strong>8</strong></td>
<td><strong>66</strong></td>
</tr>
</tbody>
</table>
5.5. Aspectizable Interfaces

<table>
<thead>
<tr>
<th>Aspectizable interfaces</th>
<th>Package or application</th>
<th>Impl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible</td>
<td>java.applet, java.awt</td>
<td>21</td>
</tr>
<tr>
<td>Cloneable</td>
<td>java.awt, java.security, java.text, java.util, JHotDraw, JGraph</td>
<td>81</td>
</tr>
<tr>
<td>Comparable</td>
<td>java.io, java.net, java.nio</td>
<td>4</td>
</tr>
<tr>
<td>Dumpable</td>
<td>FreeTTS</td>
<td>5</td>
</tr>
<tr>
<td>Externalizable</td>
<td>java.awt</td>
<td>2</td>
</tr>
<tr>
<td>Runnable</td>
<td>java.awt, java.lang, JHotDraw</td>
<td>4</td>
</tr>
<tr>
<td>Serializable</td>
<td>java.applet, java.awt, java.beans, java.io, java.security, java.lang, java.math, java.net, java.rmi, java.text, java.util, JHotDraw, JGraph</td>
<td>96</td>
</tr>
<tr>
<td>Storable</td>
<td>JHotDraw</td>
<td>35</td>
</tr>
<tr>
<td>VersionRequester</td>
<td>JHotDraw</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.17: Aspectizable interfaces found in the source code under analysis. The number of implementations is given in the last column.

mented interfaces, excluding those interfaces that are declared to be implemented, but whose implementation is completely inherited from a superclass. This means that at least one of the interface methods must be defined or redefined by the class declaring the implementation of the interface. When more than one class implements the same interface in a JDK package or in one of the three applications under analysis, the implemented interface is counted once (i.e., unique implemented interfaces are counted).

The fifth column of Table 5.16 shows the number of aspectizable interfaces. Their identification was conducted automatically and refined manually. Table 5.17 enumerates the whole list of aspectizable interfaces for JDK subpackages and for the three applications under analysis.
Refactoring of each implementation of the aspectizable interfaces produced an aspect containing the introductions necessary to add the methods required by the interface. The number of refactored implementations is given in the last column of Table 5.16. This is also the number of generated aspects (one aspect is generated per implementation). In order to remove the invocations of the interface methods, it would be possible to exploit dynamic crosscutting (pointcuts and advices). However, this was considered out of the scope of this experiment and was observed to be a minor issue for most of the code under analysis, where interface methods are seldom used in the base code itself (e.g., the methods `readObject` and `writeObject` required by the `Serializable` interface are offered to user classes, but are not used within the Java library classes themselves).

Migration of OOP code to AOP is expected to be beneficial for external quality attributes such as the understandability and maintainability, as well as for internal quality attributes, such as the modularity. In fact, the possibility to encapsulate separate concerns should result in a localized comprehension and modification effort, and an improved code structure. The empirical study described in the next section addresses the issue of the quality of the resulting aspect code.

### 5.5.3 The empirical study

Following the Goal Question Metrics (GQM) approach [2], we stated the overall aim of this study:

**Goal:** Comparison between OOP and AOP implementations of aspectizable interfaces.

The object of the study is a set of OOP systems migrated to AOP by our tool *UNPLUG*. The purpose is the comparison of the two alternative implementations, with a focus on two external quality attributes:
maintainability and understandability. The viewpoint is that of the code maintainers in a typical development setting.

5.5.4 Experimental hypotheses

The experimental hypotheses that are tested by means of this study can be expressed through the following list of research questions:

**RQ1:** Is the AOP code for the aspectizable interfaces easier to maintain than the OOP code?

**RQ2:** Is the AOP code for the aspectizable interfaces easier to understand than the OOP code?

**RQ3:** Does the migration of the aspectizable interface code produce a significant size reduction in the principal decomposition?

**RQ4:** Does the migration of the aspectizable interface code produce an improved modularity in the principal decomposition?

The research questions RQ1 and RQ2 refer to external quality attributes (understandability and maintainability), while RQ3 and RQ4 are focused on internal quality attributes (size and modularity). All of them aim at testing the hypothesis that a significant difference does hold between the AOP and the OOP implementation of the aspectizable interfaces, in terms of some external/internal quality attribute. Thus, the null hypothesis is that there is no statistically significant difference in the quality attributes for the AOP vs. OOP versions of the same applications.

RQ1 and RQ2 are the most interesting research questions, since they address quality attributes that directly impact the code evolution process. A positive answer to these questions imply that an externally measurable benefit is expected to occur when migrating the aspectizable interfaces to
AOP. On the other hand, these questions are related to quality attributes that are quite difficult to measure, in that their quantification requires the execution of a properly designed empirical study with users.

RQ3 and RQ4 are only indirectly associated with externally measurable benefits, since they refer to internal features of the source code, whose impact on the code evolution process are hypothesized but not proved. On the other hand, it is quite easy to obtain metrics that quantify these attributes, since this requires just the execution of some static code analysis.

### 5.5.5 Internal quality attributes

For the internal quality attributes, we customized some metrics available in the literature, in order to capture the size and modularity of the code in the principal decomposition.

**Metrics**

We focused on the two internal attributes, *size* and *modularity*, which are expected to be most affected by the aspectization of the crosscutting interfaces. In fact, migration to AOP of the aspectizable interfaces results in a decrease of the size of the classes that implement such interfaces. Moreover, the expected effects on the modularity are an increased cohesion of the classes, assuming that the aspectizable interface methods contribute negatively to the class cohesion, being loosely associated with the class main function. A decreased coupling is also expected, since the dependency on the aspectizable interface is removed from a class when the interface implementation is aspectized. For the definition of the modularity (cohesion and coupling) metrics, we applied the framework described in [11, 12].

*Size measures:*

**UCLOC:** Un-Commented Lines Of Code.
CHAPTER 5. ASSESSMENT  5.5. ASPECTIZABLE INTERFACES

**OP:**  Class Operations (methods, constructors, setters and getters).

In order to normalize the size measure given by the lines of code, we produce a pretty-printed version of the code without comments, in which the same formatting convention is applied to every file. The number of lines resulting from such a processing gives our UCLOC metric.

Interface aspectization results in some class properties moved to an aspect. The metric OP, counting the number of operations in each class (inherited operations excluded), allows a direct measure of the size reduction produced by the refactoring of the aspectizable interfaces.

*Modularity measures:*

**OCOH:**  Operation Cohesion: pairs of distinct methods such that the first calls the second, normalized by the maximum number of such pairs.

**ACOH:**  Attribute Cohesion: pairs of distinct methods which reference at least one common attribute, normalized by the maximum number of such pairs.

**ICOUPL:**  Interface Coupling: number of interface realization relationships.

The definition of OCOH and ACOH is motivated by the following observation: the implementation of the aspectizable interfaces is associated with transversal functions, which are not specific to the class hosting the implemented methods, so we expect that such methods are seldom used by, or using, other methods. Moreover, being not associated with the class main responsibility, these methods might operate on a small subset of the class attributes. Thus, we measure the connections between methods due to invocation or attribute reference, in order to capture the increased con-
nectivity (i.e., cohesion) that is expected to occur when the methods that implement the aspectizable interfaces are refactored.

In measuring OCOH, polymorphism is dealt with statically. In measuring ACOH, getter/setter methods and constructors are excluded, because these special purpose methods must necessarily reference the class attributes, thus introducing a bias in the cohesion metric. In measuring both OCOH and ACOH, inherited properties (methods and attributes) are excluded. OCOH and ACOH satisfy the four properties proposed in [11] for the theoretical validation of any cohesion metric (non-negativity and normalization, null value, monotonicity, merging of unconnected classes).

Since we aim at decoupling the implementation of the aspectizable interfaces from the principal decomposition, the chosen coupling metric counts the number of interfaces implemented by a class (completely inherited interface implementations are not counted). This corresponds to the intuitive notion of a class being coupled to an interface whenever the former implements the latter. ICOUPL satisfies the five properties proposed in [12] for the theoretical validation of any coupling measure (non-negativity, null value, monotonicity, merging of classes, merging of unconnected classes).

**Statistical test**

Statistical tests are used to determine whether the differences between the metrics collected either in the AOP or in the OOP setting are significant or not. For the internal (size and modularity) metrics, the Pearson Chi-square test can be used.

The Chi-square test is a non-parametric test of statistical significance for bivariate tabular analysis. It computes the degree of confidence in rejecting the null hypothesis, which states that the differences observed between two samples, taken from two populations, are due to random error. If the null hypothesis is rejected by the Chi-square test, the two samples are different.
enough, with respect to the considered characteristic, that we are allowed to generalize and conclude that the whole populations from which the samples are drawn are also different in the same characteristic. This means that the relationship between the samples and the characteristic is systematic in the populations and is not due to random error alone.

The Chi-square test requires to fix the tolerance for error, i.e., the probability of rejecting a true null hypothesis. This is commonly set to 0.05 (i.e. 5%). Data are grouped into categories that form a partition of the observed values. Percentiles are usually adopted in this phase. The frequencies of data in each category (percentile), divided by the independent variable (AOP vs. OOP), are compared to the expected frequencies, obtained regardless of the independent variable. The output of the Chi-square test is a Chi-square value. When this value is above a threshold (computed from the tolerance for error), the observed differences can be assumed to generalize, i.e., to be due to the independent variable, even in the larger populations. In such a case, the strength of the association between the independent variable (AOP vs. OOP) and the observed metrics (size or modularity) is given by the shared variance, i.e., the portion of the total variable distribution which is due to the relationship between independent and dependent variables.

5.5.6 External quality attributes

For the external quality attributes, we mapped the intuitive notions of maintainability and understandability into an effort metric.

Metrics

Quantification of the external quality attributes considered in this study requires the execution of an empirical study, involving some subjects executing some maintenance tasks either on the AOP or on the OOP code
The times measured during the execution of the maintenance tasks are used to estimate the maintenance and understanding effort. The chosen metrics for *maintainability* and *understandability* are respectively:

**MTIME:** Maintenance Time: total time measured during the execution of each maintenance task.

**UTIME:** Understanding Time: time measured during the execution of each maintenance task, in the *Understanding* state.

In order to measure the understanding time separately from the total maintenance time, a *Time recorder* tool was developed. It is displayed on the screen as a small window, that shows the state the programmer is in, during the execution of the maintenance task. A simple two-state model was adopted to distinguish between the *Understanding* state, in which the programmer is comprehending the code structure or is determining the code fragments to be changed, and the *Modifying* state, in which the programmer is editing the source code to implement the requested change, or is compiling/executing the code to see the effects of the change. State modification is triggered by the programmer, who is requested to click on a button of the *Time recorder* upon each state change. The current state of the *Time recorder* is shown as the button label.

**Experimental design**

An empirical study was designed in order to measure the maintenance and understanding effort required to complete a set of typical maintenance tasks, executed either on the AOP or on the OOP code. The experiment involves 4 groups of subjects and 2 maintenance tasks. The independent variables are the source code being used (either OOP or AOP) and the task being executed (either Task1 or Task2). The dependent variables are the maintenance time (MTIME) and the understanding time (UTIME).
Table 5.18: Experimental design of the empirical study to measure maintenance and understanding times.

In order to minimize the effect of the order in which the tasks are executed and the two code variants (OOP vs. AOP) are modified, we counterbalanced both the task execution order and the order of code modification. The resulting experimental design is shown in Table 5.18. The OOP code is the first to be modified by the groups G1 and G4, while the AOP code is modified first by G2 and G3. Moreover, Task1 is executed as the first task by the groups G1 and G2, while Task2 is executed first by G3 and G4.

It should be noted that a factorial (completely-balanced) design, involving $2 \times 4 \times 2 = 16$ sessions, is not feasible in our case, since replication of the same task by the same group under different treatments makes no sense, because re-execution of the same task would be simplified by the previously accumulated knowledge (memory effect). Thus, we had to opt for the partially-balanced design depicted in Table 5.18.

**Statistical test**

For the effort metrics collected during the execution of the empirical study the appropriate statistical test is the *Anova* (Analysis of variance).

The Anova test aims at determining if the differences in means of the dependent variables are due to random error or to the independent variables (called *factors* in this context). This is obtained by partitioning the
total variance into the component that is due to random error and the components that are due to differences between means. These latter variance components are then tested for statistical significance (via $F$-statistics), and, if significant, the null hypothesis of no differences between means is rejected.

The significance level $\alpha$ used in the $F$-statistics (probability of rejecting a true hypothesis) is usually set to 0.05. The output of the Anova test consists then of the $p$-value, i.e., the probability that the observed difference between means is due to pure chance. The commonly used significance threshold for the $p$-value is 0.05. Below this value the null hypothesis is rejected and the difference between means is considered statistically significant.

### 5.5.7 Internal quality attribute values

**Metrics**

Table 5.19 shows the frequencies of the size and modularity metrics within each quartile (including 1/4 of the observed values, considered in increasing order), evaluated on the source code, both the original and the refactored version. Only classes affected by the refactoring are considered (i.e., metrics are not computed for the generated aspects), since we are interested in evaluating the effect of migration on the principal decomposition. The crosscutting concerns, now modularized, are handled separately.

For each metric, quartiles are determined on the whole set of classes (both original and refactored classes), as apparent from row *Total*, where (approximately) the same number of classes populate each range. In the two rows above *Total* (labeled *Original* and *Refactored*) the classes in each range are distinguished based on the code provenance. The interesting case is when these two numbers differ for a given range of a metric. This
5.5. ASPECTIZABLE INTERFACES

<table>
<thead>
<tr>
<th>UCLOC</th>
<th>12-85</th>
<th>85-220</th>
<th>223-388</th>
<th>392-3408</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>47</td>
<td>47</td>
<td>54</td>
<td>49</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(23.9%)</td>
<td>(23.9%)</td>
<td>(27.4%)</td>
<td>(24.9%)</td>
<td></td>
</tr>
<tr>
<td>Refactored</td>
<td>53</td>
<td>51</td>
<td>45</td>
<td>48</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(26.9%)</td>
<td>(25.9%)</td>
<td>(22.8%)</td>
<td>(24.4%)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>98</td>
<td>99</td>
<td>97</td>
<td>394</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OP</th>
<th>1-10</th>
<th>11-20</th>
<th>21-35</th>
<th>36-252</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>45</td>
<td>51</td>
<td>51</td>
<td>50</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(22.8%)</td>
<td>(25.9%)</td>
<td>(25.9%)</td>
<td>(25.4%)</td>
<td></td>
</tr>
<tr>
<td>Refactored</td>
<td>55</td>
<td>48</td>
<td>49</td>
<td>45</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(27.9%)</td>
<td>(24.4%)</td>
<td>(24.9%)</td>
<td>(24.9%)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>99</td>
<td>100</td>
<td>95</td>
<td>394</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OCOH</th>
<th>0.0-0.0102</th>
<th>0.0103-0.021</th>
<th>0.0219-0.0413</th>
<th>0.0414-1.0</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>58</td>
<td>51</td>
<td>52</td>
<td>36</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(29.4%)</td>
<td>(29.9%)</td>
<td>(26.4%)</td>
<td>(18.3%)</td>
<td></td>
</tr>
<tr>
<td>Refactored</td>
<td>42</td>
<td>48</td>
<td>46</td>
<td>61</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(21.3%)</td>
<td>(24.4%)</td>
<td>(23.3%)</td>
<td>(31.0%)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>99</td>
<td>98</td>
<td>97</td>
<td>394</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACOH</th>
<th>0.0-0.0396</th>
<th>0.04-0.1667</th>
<th>0.17-0.3889</th>
<th>0.3928-1.0</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>54</td>
<td>49</td>
<td>51</td>
<td>43</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(27.4%)</td>
<td>(24.9%)</td>
<td>(25.9%)</td>
<td>(21.8%)</td>
<td></td>
</tr>
<tr>
<td>Refactored</td>
<td>46</td>
<td>50</td>
<td>47</td>
<td>54</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(23.4%)</td>
<td>(25.4%)</td>
<td>(23.9%)</td>
<td>(27.4%)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>99</td>
<td>98</td>
<td>97</td>
<td>394</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ICOUPL</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3-7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>0</td>
<td>79</td>
<td>74</td>
<td>44</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(0.0%)</td>
<td>(40.1%)</td>
<td>(37.6%)</td>
<td>(22.3%)</td>
<td></td>
</tr>
<tr>
<td>Refactored</td>
<td>103</td>
<td>67</td>
<td>21</td>
<td>6</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(52.3%)</td>
<td>(34.0%)</td>
<td>(10.7%)</td>
<td>(3.0%)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>103</td>
<td>146</td>
<td>95</td>
<td>50</td>
<td>394</td>
</tr>
</tbody>
</table>

Table 5.19: Internal quality metrics, divided into quartiles and computed either on the original or on the refactored code.
indicates that original and refactored code differ with respect to the given metric, in that they have a different number of classes with the metric in the given range.

Let us consider, for example, the metric OP. We can notice that 100 classes (approximately 25%) have a value of OP between 1 and 10 (first quartile), for 99 classes the value of OP is between 11 and 20 (second quartile), for 100 classes it is between 21 and 35 and for 95 classes it is between 36 and 252. Within the first quartile for OP, 45 classes (22.8%) belong to the original code, while 55 classes (27.9%) are from the code of the migrated applications. Similarly, in the 2nd, 3rd and 4th quartiles there are respectively 51, 51, 50 (25.9%, 25.9%, 25.4%) classes from the original code and 48, 49, 45 (24.4%, 24.9%, 24.9%) classes from the refactored code. For ICOUPL the first two quartiles are replaced by the sets of classes respectively with this metric equal to 0 or 1, since the value of the median (1) does not allow splitting the data into two sub-groups.

Whenever in the four quartiles a deviation of the relative frequencies can be observed with respect to the value 25%, we can hypothesize that the given metric differs in the original and in the refactored code. For example, less than 25% of the original classes have a value of OP in the 1st quartile, while more than 25% of the original classes have a higher value of OP (either in the 2nd, 3rd or 4th quartiles). Correspondingly, the refactored code falls in the 1st quartile in more than 25% of the cases, while it is in the other quartiles with a relative frequency lower than 25%. This seems to indicate a trend: in the refactored code the metric OP tends to be lower, since the 1st quartile contains more refactored than original classes. A similar trend can be observed for UCLOC and ICOUPL, while the opposite trend (higher value in the refactored code) can be observed for OCOH and ACOH. This would indicate a lower size, a lower coupling and a higher cohesion of the principal decomposition code after migrating
CHAPTER 5. ASSESSMENT

5.5. ASPECTIZABLE INTERFACES

<table>
<thead>
<tr>
<th>Metric</th>
<th>Threshold</th>
<th>Chi-square</th>
<th>Shared var.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCLOC</td>
<td>7.82</td>
<td>1.35</td>
<td>-</td>
</tr>
<tr>
<td>OP</td>
<td>7.82</td>
<td>1.39</td>
<td>-</td>
</tr>
<tr>
<td>OCOH</td>
<td>7.82</td>
<td>9.46</td>
<td>2.4%</td>
</tr>
<tr>
<td>ACOH</td>
<td>7.82</td>
<td>2.06</td>
<td>-</td>
</tr>
<tr>
<td>ICOUPL</td>
<td>7.82</td>
<td>162.43</td>
<td>41.2%</td>
</tr>
</tbody>
</table>

Table 5.20: Chi-square thresholds and values. For statistical significance, Chi-square > Threshold is required.

the aspectizable interfaces. However, this conjecture must be subjected to the Chi-square test to check its statistical significance.

Statistical analysis

Table 5.20 shows the Chi-square figures compared with the respective thresholds for all the computed metrics. When the Chi-square computed on the sample data is greater than the threshold, the null hypothesis can be rejected. We achieve significance only for OCOH and ICOUPL, while for the other metrics the different distribution across the quartiles is not statistically significant.

If the null hypothesis can be rejected, we conclude that there exists a statistically significant relationship between the dependent variables (the metrics) and the independent (original vs. refactored code) variable. The shared variance, reported in Table 5.20 for the two metrics (OCOH and ICOUPL) that differ in a significant way, measures the strength of such a relationship. Thus, 2.4% of the variance of OCOH and 41.2% of the variance of ICOUPL are due to the difference between original and refactored code. Migration of the aspectizable interfaces to aspects produced a significantly increased cohesion of the operations in each class (OCOH) and an even more significant decrease of the coupling with the interfaces (ICOUPL). On the other hand, no significant size reduction (metrics
5.5. ASPECTIZABLE INTERFACES

Figure 5.4: Class diagram for the package java.util, with interfaces migrated to aspects in bold boxes (a). At the bottom (b), class diagram (principal decomposition) after migration of the aspectizable interfaces.

UCLOC and OP) and no significant increase of the cohesion around the attributes (ACOH) was observed. Thus, original and refactored code have comparable size and comparable cohesion of the classes around the class attributes.

The effect of the decrease of ICOUPL on the understandability of the class diagram is apparent when Figure 5.4 (a) is compared to Figure 5.4 (b). The hierarchy of the classes implementing the Collection interface, as well as that related to the Map interface, can be immediately identified in the new view. The diagram is no longer cluttered by crosscutting concerns, which previously obscured the core hierarchies and can now be represented in a separated view (aspect diagram), as shown in Figure 5.5. In this diagram, the presence of three super-aspects (AbstractSerializable_Map, AbstractSerializable_Entry, AbstractCloneable) indicates that some implementations of the methods required by the aspectizable interfaces (Serializable and Cloneable) were the same. In the refactored version, such code is no longer duplicated, being inherited from the parent aspect.
The increased OCOH can be explained by the fact that the methods introduced to implement an aspectizable interface are often only loosely coupled with the principal functions operated in a given class. Correspondingly, they are seldom called by the other methods of the same class. For example, the methods `readObject` and `writeObject`, required to implement the `Serializable` interface, are called only implicitly when an object is serialized or deserialized to/from and object stream. No method of the class they belong to usually calls them directly. Separation of the methods which do not contribute to the main class functions improves the conceptual integrity of the class and potentially simplifies understanding.
5.5.8 External quality attribute values

Maintenance tasks

The two tasks we defined for the empirical study are related to the source code of the `java.awt` package (and its sub-packages), from JDK. Here is a summary description:

Task1: In all the implementations of the `Cloneable` interface, a coding rule (not respected consistently in the `java.awt` package) on the scope of the variable declarations has to be enforced. It requires that variables be declared within the smallest scope where they are used.

Task2: A specific error reporting protocol has to be enforced whenever the implementation of the `clone()` method, required by the `Cloneable` interface, invokes another implementation of `clone()` (either from the superclass or from the class of an attribute). All code portions not respecting the protocol must be updated to implement it.

The following criteria have been used to define them: tasks should be representative of typical interventions that can be required for library code such as that of JDK. They should have an intermediate complexity, making them non trivial to accomplish. At the same time, it should be possible to complete them within a reasonable time, since subjects are available for a limited time slot. We gave the subjects a limit of 2 hours and they were always able to complete them within such time.

Subjects

Twelve subjects were involved in the 24 experimental sessions conducted in our empirical study. Five subjects are researchers, four are programmers, three are students (two graduate and one undergraduate). All of them had
previous experience with Java and OOP. All were already familiar with 
JDK. In order to evaluate their skills in Java and AspectJ programming,
subjects’ were administered an exercise on the first day of the experiment,
after the tutorial on AspectJ. Successful completion of the exercise was con-
sidered an indicator of sufficient skill for the tasks at hand. Subjects were 
divided into four groups (G1-G4), each containing 3 subjects, and were 
randomly assigned Task1/Task2 in the OOP vs. AOP setting according to 
the scheme shown in Table 5.18.

Setting

The maintenance tasks were executed in a typical programming environ-
ment, consisting of a code editor of choice (e.g., Emacs) and either the 
Java compiler javac or the AspectJ compiler ajc. The HTML documenta-
tion generated by means of javadoc was made available, in addition to the 
source code.

On the first day of the experiment, all subjects were given a short tuto-
rial on AOP in general and AspectJ in particular. During such a presenta-
tion they were also given some background information on the aspectizable 
interfaces and how they had been migrated to aspects. During the tutorial, 
the subjects practiced the usage of the time recorder.

On the second and third day, subjects executed the two maintenance 
task, either on AOP or on OOP code, according to the scheme prescribed by 
the experimental design. A complete environment, consisting of the source 
code, the HTML documentation and a compilation script, was installed on 
their machine.

After the completion of each experimental session, we collected the times 
measured by the time recorder, we made a copy of the modified source code 
to check the successful completion of the tasks, and we gathered some free-
format comments by the subjects, about the difficulties they encountered
Table 5.21: Understanding time (UTIME) and total maintenance time (MTIME) in seconds, measured in each experimental session. Average (Avg) and variance (Var) are also shown.

and the way they proceeded.

**Metrics**

Table 5.21 reports the amount of time (in seconds) spent in the execution of the maintenance tasks (both total time, MTIME, and understanding time, UTIME). Apparently, a reduction of the times in the AOP setting can be observed with respect to OOP, if we look at the averages. However, a high variability is also apparent from the values of the variances (in Table 5.21, \( Var = \sum_{i=1}^{n} (x_i - \bar{x})^2/(n - 1) \)). Thus, execution of the Anova test
Table 5.22: Two-factor Anova for MTIME and UTIME. SS = Sum of Squares, df = degrees of freedom, MS = Mean sum of Squares. For statistical significance, F > F-crit is required.

<table>
<thead>
<tr>
<th>Source of Var.</th>
<th>SS</th>
<th>df</th>
<th>MS</th>
<th>F (F-crit.)</th>
<th>P-val.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MTIME</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>7761163</td>
<td>1</td>
<td>7761163</td>
<td>26.7 (4.35)</td>
<td>0.0000466</td>
</tr>
<tr>
<td>AOP vs. OOP</td>
<td>1247616</td>
<td>1</td>
<td>1247616</td>
<td>4.30 (4.35)</td>
<td>0.0513</td>
</tr>
<tr>
<td>Interaction</td>
<td>297483</td>
<td>1</td>
<td>297483</td>
<td>1.02 (4.35)</td>
<td>0.324</td>
</tr>
<tr>
<td>Within group</td>
<td>5808465</td>
<td>20</td>
<td>290423</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UTIME</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>3164634</td>
<td>1</td>
<td>3164634</td>
<td>20.6 (4.35)</td>
<td>0.000198</td>
</tr>
<tr>
<td>AOP vs. OOP</td>
<td>1790334</td>
<td>1</td>
<td>1790334</td>
<td>11.7 (4.35)</td>
<td>0.00273</td>
</tr>
<tr>
<td>Interaction</td>
<td>696663</td>
<td>1</td>
<td>696663</td>
<td>4.54 (4.35)</td>
<td>0.0456</td>
</tr>
<tr>
<td>Within group</td>
<td>3066398</td>
<td>20</td>
<td>153320</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

is necessary to see if the difference between the averages is statistically meaningful or if it is due to the random error, which is responsible for the high variances.

Task 1 applies to the classes belonging to `java.awt`, sub-packages excluded, while Task 2 refers to `java.awt` and its subpackages. Correspondingly, subjects were given 106 files for Task 1 and 315 files for Task 2, in the OOP setting. They were given 157 files for Task 1 and 401 files for Task 2 in the AOP setting, due to the presence of the files generated for the aspects. In both settings, execution of Task 1 required the modification of 1 file (2 lines of code), while Task 2 required changing 7 files (18 lines of code).

**Statistical analysis**

Table 5.22 shows the output of the Anova test, executed on the data obtained from the empirical study (both MTIME and UTIME). Row Task
shows that there clearly is a statistically significant difference between the two tasks (\( p \)-value largely below 0.05 for both MTIME and UTIME). More interesting is row AOP vs. OOP, indicating that the understanding time is significantly lower in the AOP setting than in the OOP setting, the difference being not due to random error (null hypothesis rejected). Rows Interaction and Within group account for the proportion of the total variance respectively due to the interaction between the two factors and to the variability internal to each group.

As far as the maintenance time is concerned, although the \( p \)-value is not below the conventional threshold (0.05), it can be noticed that it is very close to such a value. We can say that the difference between maintaining the AOP vs. the OOP code versions is close to significance, from a statistical point of view. In practice, a difference is observed and the probability that such a difference is due to chance is quite low (slightly above 5%). Thus, we can conclude that aspectization of the aspectizable interface implementations results in a significantly lower understanding time, during software maintenance, and that the overall maintenance time is also quite likely to decrease, thanks to the refactoring.

**Threats to validity**

We did our best to minimize the threats to the internal validity of the study. However, the specific background of each of the involved subjects might have played a role. For example, the level of familiarity with Java and JDK in particular, and the previous experience in executing comparable maintenance tasks, might have had an impact on the difficulties encountered. We mitigated this threat to validity by replicating the study as many times as possible (24 sessions were conducted in total). The tasks chosen for the experiment may also have affected the results, although we tried to simulate *typical* maintenance interventions. Replication of the experiment
with different tasks would be necessary to address this threat. The time
recorder, which is potentially a source of disturbance, was not judged so
by the subjects, according to the comments collected after each session.

Possible threats to the external validity are the inclusion of non pro-
essional programmers in the sample and the difficulty of extrapolating to
aspects different from the aspectizable interfaces considered in this study.

5.5.9 Discussion

With regards to the internal quality attributes, aspectization of the cross-
cutting interfaces increases significantly the class cohesion associated with
the method invocations, since the extracted methods are loosely connected
with the other class operations. An even more significant decrease of the
coupling with the implemented interfaces can be observed. This is an ex-
pected result, since we are migrating those interfaces. The simplification of
the class diagram associated with the reduced coupling with the interfaces,
combined with the increased class cohesion, might explain the improved
understandability that was observed in the execution of the maintenance
tasks.

The main result of the user study is that the migration of the aspec-
tizable interfaces produces code that is easier to understand, during the
execution of a maintenance task. The overall effect on the maintainability
is somehow less apparent. A possible explanation for this, which would
require further empirical validation, is that refactoring of the aspectizable
interfaces does not affect significantly the principal decomposition size, as
resulting from the related metric, so that the modification phase is only
marginally improved.

Let us reconsider the initial research questions. The answer to RQ1 is
partially positive (the AOP code is quite likely to be easier to maintain).
The answer to RQ2 (on the code understandability) is definitively positive.
No significant size reduction was observed, associated with the migration of the aspectizable interfaces (RQ3), while the modularity metrics for the class cohesion due to method invocations and for the class coupling with the interfaces were significantly improved (resp. increased and decreased), so that RQ4 can be given a positive answer.

Overall, this study indicates that migration of the aspectizable interfaces has a limited impact on the principal decomposition size, but at the same time it produces an improvement of the code modularity. From the point of view of the external quality attributes, modularization of the implementation of the crosscutting interfaces clearly simplifies the comprehension of the source code. We hypothesize that further benefits in the overall maintainability would be achieved if a larger fraction of the code were affected by the migration to AOP. However, further experiments are necessary to validate this hypothesis.
Chapter 6

Conclusions and Future Works

6.1 Conclusions

The topic of the research presented in this dissertation is how to support the transformation of an existing software system written using a traditional (OOP) programming paradigm into an equivalent one that takes advantage of the improved separation of concerns provided by aspect oriented programming languages. The goal of this code change is to remove the drawbacks due to the presence of crosscutting concerns. Traditional languages fail in modularizing functionalities that are tangled and scattered with respect to the principal decomposition offered by the language in use. Correspondingly the source code is hard to understand, to maintain and to evolve, because the comprehensibility of the principal system decomposition is complicated by the presence of these tangled crosscutting concerns. Moreover, the comprehensibility of the crosscutting concern itself can not be so easy, since it is scattered throughout a big portion of the system. AOP provides a better modularization of the source code, because scattered concerns can be now implemented into a single place, an aspect. In this way, many comprehensibility problems due to tangling and scattering can be addressed.

Migration towards AOP has been broken down into two steps. Aspect
mining is the first step. It consists of the identification of the crosscutting concerns in the existing application. These crosscutting concerns are the aspect candidates. After identification, the actual transformation takes place in the refactoring step.

The proposed techniques have been implemented in a collection of tool prototypes which are freely available. The proposed methods have been applied to a number of software systems for a total of more than half a million lines of code, in order to show the feasibility of the migration process and to evaluate the results.

Dynamic Aspect Mining

We support the identification of aspect candidates in existing systems by means of a semi-automated method, which has been implement in a tool prototype, Dynamo. The proposed aspect mining method involves running a set of use-cases, applying concept analysis and isolating use-case specific concepts whose attributes satisfy the scattering and tangling constraints. The required use-cases do not need to be complete with respect to the application’s requirements. Rather, they are supposed to exercise both base and crosscutting functionalities. Their selection should be driven by knowledge about the functionalities provided by the given system, regarded as a black box. No intimate knowledge of the implementation is assumed. The technique can automatically separate well modularized functionalities that are exercised by some use-cases, from those functionalities that cannot be assigned to a single modular units. The latter are reported as aspect candidates and are submitted to the user for (manual) interpretation.

We applied the proposed technique to a case study consisting of about 40,000 lines of code, JHotDraw. Several of the aspect candidates reported by the proposed technique can be regarded as potentially good aspects, either because they are mentioned as typical examples in the AOP literature
or because a manual evaluation of the results confirmed their meaningfulness. For example, Persistence, Observer and Visitor are often presented as classical examples where the AOP implementation is superior to the traditional one. Undo and Manage Handles are examples of aspect candidates reported by our technique, that are specific of the given application and thus are not mentioned in AOP text books. We conducted a preliminary assessment of the accuracy of the results for a selected concern (Undo). Although some manual adjustment is necessary, especially to complete the list of methods reported as aspect seeds, the starting point that is produced automatically by the proposed technique is a valuable one. The need for a completion of the reported seeds is somehow expected, since we are using a dynamic method, which is by definition partial. This opens to the possibility of interesting combinations with other static methods.

Comparison and combination with other techniques

We compared dynamic aspect mining with two other aspect mining techniques, fan-in analysis and identifier analysis. We assessed their respective strengths and weakness by applying them to a common benchmark application, JHotDraw. A combined brand new technique has also been developed, based on this assessment.

We observed that all three techniques are able to identify seeds for well-known crosscutting concerns, but that interesting differences arose for other concerns. These differences are largely due to the different ways in which the techniques work. Fan-in analysis is good at identifying seeds that are largely scattered throughout the system and that involve a lot of invocations of the same method, but it cannot be used to analyze smaller applications. Identifier analysis is able to identify seeds when the associated methods have low fan-in, but only if these methods share a common lexicon. The main drawback of this technique is the large number of reported seeds
that have to be inspected manually. Finally, dynamic aspect mining is able to find seeds in the absence of high fan-in values and common identifiers, but the technique is only partial because it relies on execution traces.

We also observed that the three techniques are quite complementary: fan-in analysis and dynamic aspect mining require a manual effort to expand the seeds into full concerns, whereas identifier analysis covers a large part of a concern, but requires extensive filtering of the reported seeds. Hence, to improve automation of both fan-in analysis and dynamic aspect mining, and to reduce the search space for identifier analysis, we proposed a combined technique in which seeds from either fan-in analysis or dynamic aspect mining are expanded automatically by applying identifier analysis. To verify the performance of this combined technique, we applied it to JHotDraw and interpreted the results in terms of two indicators: precision and recalled methods. The measures show that for three out of the four concerns we considered, the combined technique outperforms the individual techniques. In only one case, the combined technique performed worse.

**Pointcut extraction**

A semi-automated approach has been presented to support the migration from OOP code to AOP code. In particular, the applicability of semantic-preserving code refactoring transformations to automate the migration task was considered. Given a source program with the aspectual fragments marked, the transformation produces a semantically equivalent program with the marked fragments migrated to aspects.

A tool for OOP to AOP migration has been implemented, AOP-Migrator, which implements several variant of six principal refactorings, and illustrated its application as part of a migration strategy on four medium-sized case studies, for a total of 82,700 lines of code. The results show that it is possible to migrate from OOP to AOP in a largely automated way. The
refactorings are entirely automated. However, they are not always directly applicable, necessitating the application of enabling transformations which map OO programs into equivalent OO programs. These enabling transformations also can be largely automated.

The goal to completely automate the migration from OOP to AOP is probably as undesirable as it is unachievable; there are some refactoring decisions which are best left to the human. There are also situations where, as with other transformation based work, the human is best placed to determine the sequence of enabling transformations that will allow the application of a chosen refactoring.

However, as the case studies illustrate, many of the enabling transformation steps can be automated; they rely upon standard, well-understood transformation steps, such as side effect removal and statement re-ordering. A preliminary experiment on the generated aspect code suggests those pointcuts that are used by similar advices can be easily grouped together and substituted by a more abstract and more easy to understand pointcut.

The approach do not make any assumption on the required aspect mining phase, which can be performed either manually or with the aid of any of the automatic/semi-automatic methods proposed in literature. When combined with existing (and automated) approaches to aspect identification, tool-supported migration from OOP to AOP is achieved.

Aspectizable interfaces

The notion of aspectizable interfaces has been defined and described. Some heuristics have also been defined and implemented to automatically identify which interface implementations match this definition. Even if simple, these methods performed quite well on an initial test base, producing very good results in terms of precision and recall.

After a first smaller experiment, used to evaluate the feasibility of au-
tomatically identifying aspectizable interfaces, a larger number of classes (about 1,300 classes) was subjected to the technique for the migration of the aspectizable interfaces to aspects. We assessed the resulting code in terms of its internal and external quality attributes through an empirical study. The results indicate that among the internal attributes, only those referred to the modularity had a significant change. The size was not significantly affected by the migration. This is due to the fact that the amount of code devoted to the implementation of the aspectizable interface methods is typically a small fraction of the overall size.

Externally, the refactored code, which resorts to aspects for the implementation of the aspectizable interfaces, is definitively easier to understand than the code that mixes such implementation with the primary class responsibilities. To some extent, it is also easier to maintain, although the small size of the affected code portion might, in our opinion, have reduced the overall benefits on maintainability.

Clearly, it is difficult to extrapolate from a single study, focused on a specific kind of aspects – the crosscutting interface implementation. However, the strong indication of an improved understandability represents an important result for the AOP research at large. This is the first empirical study with users, in which an external quality attribute of the source code is shown to be improved by the migration to aspects.

6.2 Future work

The results of dynamic aspect mining are quite promising and represent a provisional validation of the proposed method. However, several open issues must be considered in the future work. For example, the granularity of the use-cases might affect the quality of the resulting concept lattice.

In our future work we will extend the proposed comparison framework
with other aspect mining techniques, which could give rise to new interesting combinations of such techniques. This will not only allow us to come up with better (combined) aspect mining techniques, but will also allow us to evaluate the considered techniques even better, as new concerns will be identified that we are not currently aware of. Additionally, we could come up with extra quality indicators that complement the precision and recalled methods indicators, and empirically establish their validity by considering other benchmark applications as well.

With regards to refactoring, future work will explore the possibility of defining fully automated ‘transformation tactics’, which will allow the user to specify the goals of migration from OOP to AOP at a higher level of abstraction. Among the other issues that are left open by the present work and that we intend to investigate in the future are the development and experimentation with more advanced pointcut abstraction techniques.

As far as the empirical study an the aspectizable interfaces is concerned, we intend to replicate it in different, alternative settings. We will consider aspects often reported in the literature (e.g., persistence, logging, tracing) and see if their aspectization produces effects similar to those observed for the aspectizable interfaces. In particular, we are interested in the relationship between internal code attributes and the externally visible effects on the code maintainability. It would be interesting to refactor an increasing number of aspects, possibly of different kinds, up to the extreme situation where the principal decomposition remains empty, falling into the Multi-Dimensional Separation of Concerns paradigm [60]. Assessment of the effects on the internal and external quality attributes of the resulting code would provide useful indications about the trade-off between modules in the principal decomposition and modules for the crosscutting functionalities, thus contributing to a clarification of the notion of aspect itself.
Bibliography


olution (LATE 2005), Workshop held in conjunction with AOSD 2005, March 2005.


[65] Tom Tourwe, Andy Kellens, Wim Vanderperren, and Frederik Vanhniewenhuysse. Inductively generated pointcuts to support refactoring


Appendix A

Abstract Refactoring Descriptions

This Appendix provides an abstract description of the refactorings presented by example in Chapter 4. For each refactoring, it shows the mechanics using the syntactic classes of the Java/AspectJ languages. Variants and applicability conditions are described textually.

The notation adopted to show the mechanics of the refactorings uses the following conventions: Grammar terminals are in true-type font (e.g., class). Syntactic classes (non-terminals) are in italics (e.g., fds for class fields) and are subscripted so as to make them uniquely referable in the right part of the transformation. Their meaning is provided in the Legend underneath each refactoring. Newly generated code is in bold-italics. Transformation of the original context into the exposed context is indicated by the apex (e.g., the variable identifier vid₁ turned into vid₁'), with the rules for context exposure underneath each figure. The code to be aspectized is underlined.

The context exposed by each refactoring is constrained to all and only variables that are referenced in the code fragment being aspectized. To express that, function RefVar, which gives the set of referenced variables in a given code fragment, is used.

Fig. A.1 shows the mechanics of Extract End of Method. The call to be
class cid₁ {  
  fds₁ mds₁  
  mods₁ tp₁ mid₁ (pms₁) {  
    lvars₁ stmts₁  
    vid₁.mid₂(acts₁);  
  }  
}  

⇒  

aspect aid₁ {  
  fds₂  
  pointcut pid₁(cid₁ This, pms₁):  
    execution (tp₁ cid₁.mid₁(pmtps₁)) &&  
      this(This) && args(pmids₁);  
  after (cid₁ This, pms₁): pid₁(This, pmids₁)  
    { vid₁.mid₂(acts₁'); }  
}  

Legend:  
acts = actual parameters (sequence of vids);  
aid = aspect identifier;  
cid = class identifier;  
fds = field declarations;  
lvars = local variables;  
mds = method definitions;  
mid = method identifier;  
mods = modifier list;  
pms = formal parameters;  
pmids₁ = formal parameter identifiers of method mid₁;  
pmtps₁ = formal parameter types of method mid₁;  
stmts = statement sequence;  
tp = return type;  
vid = variable identifier.  

Exposed context:  
for each vid ∈ RefVar("vid₁.mid₂(acts₁)")  
vid = this ⇒ vid’ = This;  
vid = p ∈ pms₁ ⇒ vid’ = p;  
vid = f ∈ fds₁ ⇒ vid’ = This.f;  
vid = x ∈ lvars₁ ⇒ vid’ = x ∈ fds₂.  

Figure A.1: Mechanics for refactoring: Extract End of Method.
aspectized is at the end of the method. The related join point is intercepted by the newly generated pointcut $pid_1$, which exposes the `this` reference as well as all arguments of the enclosing method (`args` construct). An after-advice restores the original invocation, using the exposed context for the call target and for the actual parameters. When local variables are involved in the call, a copy must be kept in the aspect (creating new aspect fields, added to $fds_2$). The idiom described in Chapter 4 (with reference to the refactoring `Extract After Call`) is used in this case.

The main applicability condition for this refactoring is that every return statement must be preceded by the same code fragment to be aspectized, or, equivalently, that a single exit point exists in the method. The variant of this refactoring with the call to be aspectized at the beginning requires using a `before` advice. The variant with the call at the beginning/end of an exception handling block requires adding the primitive pointcut `handler` to $pid_1$.

Fig. A.2 shows the mechanics of `Extract After Call`. The exposed context includes the current and the target objects (`This` and `Target`) and the arguments of the called method. Local variables are exposed by copying their values into aspect fields (see Chapter 4), if necessary. In addition to matching the pattern on the left in Fig. A.2 and granting access to the context required to perform the aspectized call, a further applicability condition must be verified: the call used as a join point in the base code ($cid_2.mid_2$) must be present exactly once or must be always followed by the call being aspectized. For the variant `Extract Before Call`, when the marked block precedes the intercepted call, a before-advice is used.

Fig. A.3 shows the mechanics of `Extract Conditional`. At the top level of method $cid_1.mid_1$ a conditional statement surrounds the code to be aspectized. The base code is in the else-branch. The extracted pointcut includes a dynamic check of the aspectized condition (`if (vid_1')`). Ap-
plicability conditions for this refactoring are the presence of the syntactic pattern shown in Fig. A.3 and the possibility to expose the context required by the aspectized invocation (target object and actual parameters) as well as the context required by the condition \(vid_1\). Variants include the possibility of exchanged then- and else-branch (the pointcut condition must be negated in this case) and the possibility for the base code to be at the top level (not inside the conditional statement) in method \(cid_1.mid_1\). In the latter case, the `proceed` construct is used to ensure the execution of the base code in the case of the aspect advice activation.

Fig. A.4 shows the mechanics of `Pre Return`. The around advice that replaces the original execution stores the return value, obtained through the `proceed` construct, into a temporary variable, which is returned only after performing the aspectized invocation. The temporary variable is exposed according to the normal context exposure rules, unless it is a local variable of \(mid_1\). In such a case, it is declared locally in the advice body (with type \(tp_1\)). It should be noticed that the pattern in Fig. A.4 includes the main applicability condition for this refactoring: the expression in the return statement must consist of a single variable \(vid_2\). When the input code does not match this condition, it is relatively easy to transform the code so as to satisfy it (e.g., by introducing a local variable which holds the value to be returned). Moreover, a single return point is assumed.

Fig. A.5 shows the mechanics of `Extract Wrapper`. One of the parameters of the call to \(cid_4.mid_2\) is an object of class \(cid_2\) that wraps another object, of class \(cid_1\). In the transformed code, the aspect intercepts the call to \(cid_4.mid_2\). It exposes the un-wrapped parameter \(vid_3\) and continues the original call (via `proceed`) with a new wrapper object \(vid_2\) replacing the original parameter. Since the join point intercepted by the aspect is a call to \(cid_4.mid_2\), the applicability condition is the same as for `Extract After Call`. That is, just one such call must be present or, in the presence of
multiple calls, each of them must have a wrapper object as a parameter, with all wrappers built in the same way. As a variant, it is possible to intercept the construction of $vid_1$, instead of the call to $mid_2$ (as discussed in Chapter 4).

Finally, Fig. A.6 shows the mechanics of Extract Exception Handling. The catch block for exceptions of type $tp_2$ is removed from the base method body, thanks to the declare soft construct in the aspect, which softens such exceptions. When a $SoftException$ is raised in the execution of the original method, an after-advice is triggered in the aspect, which runs the original exception handling code (using the exposed context to reference the needed locations). If $stmts_2$ does not end with a throw or an exit statement, it is necessary to use an around advice instead of an after advice, so as to avoid throwing a $SoftException$. 
class cid₁ {
  fds₁ mds₁
  mods₁ tp₁ mid₁ (pms₁) {
    lvars₁ stmts₁
    vid₁,mid₂(acts₁);
    stmts₂
  }
}

aspect aid₁ {
  fds₂
  pointcut pid₁(cid₁ This, cid₂ Target, pms₂):
    withincode (tp₁ cid₁,mid₁(pmtps₁) & &
    call (tp₂ cid₂,mid₂(pmtps₂) & &
     this(This) & &
     target(Target) & & args(pmids₂));
  after (cid₁ This, cid₂ Target, pms₂):
    pid₁(This, Target, pmids₂)
    { vid₂'.mid₃(acts₂'); }  
}

Legend:
acts = actual parameters (sequence of vids);
aid = aspect identifier;
cid = class identifier;
cid₂ = class of method mid₂;
fds = field declarations;
lvars = local variables;
mds = method definitions;
mid = method identifier;
mods = modifier list;
pms = formal parameters;
pms₂ = formal parameters of method mid₂;
pmidts₂ = formal parameter identifiers of method mid₂;
pmtps₁ = formal parameter types of method mid₁;
pmtps₂ = formal parameter types of method mid₂;
stmts = statement sequence;
 tp = return type;
 tp₂ = return type of method mid₂;
vid = variable identifier.

Exposed context:
for each vid ∈ RefVar(“vid₂,mid₃(acts₂)”)  
vid = this ⇒ vid’ = This;
vid = f ∈ fds₁ ⇒ vid’ = This.f;
vid = x ∈ lvars₁ ⇒ vid’ = x ∈ fds₂;
vid = vid₁ ⇒ vid’ = Target.

Figure A.2: Mechanics for refactoring: Extract After Call.
APPENDIX A. ABSTRACT REFACTORIZING DESCRIPTIONS

```
class cid1 {
    fds1 mds1
    mds1 tp1 mid1 (pms1) {
        if (vid1) {
            vid2,mid2(acts1);
        } else {
            stmts1
        }
    }
}

aspect aid1 {
    fds2
    pointcut pid1(cid1 This, pms1):
        execution(tp1 cid1,mid1(pmtps1)) &&
        this(This) && args(pmids1) && if (vid1');
    void around (cid1 This, pms1):
        pid1(This, pmids1)
        { vid2'.mid2(acts1'); }
}
```

Legend:
acts = actual parameters (sequence of vids);
aid = aspect identifier;
cid = class identifier;
fds = field declarations;
lvars = local variables;
mds = method definitions;
mid = method identifier;
mods = modifier list;
pms = formal parameters;
pmids1 = formal parameter identifiers of method mid1;
pmtps1 = formal parameter types of method mid1;
stmts = statement sequence;
sp = return type;
vid = variable identifier.

Exposed context:
for each vid ∈ RefVar("vid2,mid2(acts1)"")
vid = this ⇒ vid' = This;
vid = p ∈ pms1 ⇒ vid' = p;
vid = f ∈ fds1 ⇒ vid' = This.f;
vid = x ∈ lvars1 ⇒ vid' = x ∈ fds2.

Figure A.3: Mechanics for refactoring: Extract Conditional.
APPENDIX A. ABSTRACT REFACTOING DESCRIPTIONS

Figure A.4: Mechanics for refactoring: Pre Return.

```
class cid1 {
    fds1 mds1
    mods1 tp1 mid1 (pms1) {
        lvars1 stmts1
        return vid2
    }
}

aspect aid1 {
    fds2
    pointcut pid1(cid1 This, pms1):
        execution (tp1 cid1,mid1(pmtps1)) &&
                    this(This) && args(pmids1);
    tp1 around (cid1 This, pms1): pid1(This, pmids1) {
        tp1' vid2' = proceed(This, pmids1);
        vid1'.mid2(acts1');
        return vid2'
    }
}
```

Legend:
acts = actual parameters (sequence of vids);
aid = aspect identifier;
cid = class identifier;
fds = field declarations;
lvars = local variables;
mds = method definitions;
mid = method identifier;
mods = modifier list;
pms = formal parameters;
pmids1 = formal parameter identifiers of method mid1;
pmtps1 = formal parameter types of method mid1;
stmts = statement sequence;
tp = return type;
vid = variable identifier.

Exposed context:
for each $vid \in \text{RefVar}(\"vid1,mid2(acts1)\") \cup \{vid2\}$
vid = this $\Rightarrow$ vid' = This;
vid = p $\in$ pms1 $\Rightarrow$ vid' = p;
vid = f $\in$ fds1 $\Rightarrow$ vid' = This.f;
vid = x $\in$ lvars1 $\Rightarrow$ vid' = x $\in$ fds2.

$tp1' = \epsilon$ (empty string)
if vid = vid2 $\in$ lvars1 $\Rightarrow$ vid2' = vid2

$tp1' = tp1$
interface iid1 { fds1 mdecls1 }
class cid1 implements iid1 {
    fds2 mds1 }
class cid2 implements iid1 {
    fds3 mds2 }
class cid3 {
    fds4 mds3
    mods1 tp1 mid1 (pms1) {
        lvars1 stmts1
        cid1 vid1 = new cid1(acts1);
        cid2 vid2 = new cid2(vid1, acts2);
        vid3.mid2(vid2, acts3);
        stmts2
    }
}

aspect aid1 {
    fds2
    pointcut pid1(cid3 This, pms2):
        withincode(tp1 cid3.mid1(pmtps1) &&
            call(tp2 cid4.mid2(pmtps2)) &&
            this(This) && arg(pmids2);
    void around((cid3 This, pms2):
        pid1(This, vid3, pmids3) {
            cid2 vid2 = new cid2((cid1)vid3, acts2);
            proceed(This, vid2, pmids3);
        }
    }
}

Exposed context:
for each vid ∈ RefVar("acts2")
vid = this ⇒ vid' = This;
vid = f ∈ fds1 ⇒ vid' = This.f;
vid = x ∈ lvars1 ⇒ vid' = x ∈ fds2.

Figure A.5: Mechanics for refactoring: Extract Wrapper.
APPENDIX A. ABSTRACT REFACTORING DESCRIPTIONS

The figure illustrates the mechanics for refactoring to extract exception handling. The code snippet shows the transformation of a class with exception handling to an aspect that declares a soft type and applies it within the original class.

```java
class cid1 {
    fds1 mds1
    mods1 tp1 mid1 (pms1) {
        try {
            lvars1 stmts1
        } catch(tp2 vid1) {
            stmts2
        }
    }
}

import org.aspectj.lang.SoftException;
aspect aid1 {
    declare soft: tp2:
        ( call(*.*(..) throws tp2) ||
        call(*.new(..) throws tp2) ) &&
        withincode(tp1 cid1.mid1(pmtps1));
    after(cid1 This, pms1)
    throwing(SoftException vid1):
        execution(tp1 cid1.mid1(pmtps1)) &&
        this(This) && args(pmids1) {
            stmts2'
        }
}
```

Legend:
- **acts** = actual parameters (sequence of vids);
- **aid** = aspect identifier;
- **cid** = class identifier;
- **fds** = field declarations;
- **lvars** = local variables;
- **mds** = method definitions;
- **mid** = method identifier;
- **mods** = modifier list;
- **pms** = formal parameters;
- **pmids1** = formal parameter identifiers of method mid1;
- **pmtps1** = formal parameter types of method mid1;
- **stmts** = statement sequence;
- **tp** = return type;
- **vid** = variable identifier.

Exposed context:
- For each vid ∈ RefVar("stmts2")
  - vid = this  ⇒  vid' = This;
  - vid = p ∈ pms1  ⇒  vid' = p;
  - vid = f ∈ fds1  ⇒  vid' = This.f;
  - vid = x ∈ lvars1  ⇒  vid' = x ∈ fds2.

Figure A.6: Mechanics for refactoring: Extract Exception Handling.