MIGUEL JORGE TAVARES PESSOA MONTEIRO

Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts

Tese de Doutoramento em Informática

Tese submetida à Universidade do Minho, para a obtenção do grau de Doutor,
elaborada sob a orientação do Professor Doutor João Miguel Lobo Fernandes

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Braga, Março 2005

ABSTRACT

Software engineering tools should support complete separation of concerns, by enabling the deployment of each different concern in its own unit of modularity. Unfortunately, current tools and languages – including those supporting the object-oriented programming paradigm – fail to provide a complete and effective support for the separation of all concerns. Undesirable phenomena such as code scattering and code tangling ensue.

Aspect-Oriented Programming is a new programming paradigm capable of modularising crosscutting concerns. Aspect-oriented programming complements existing programming paradigms, including object-oriented programming, with constructs that provide a fuller separation of concerns.

Refactoring is a technique to restructure program source code in order to improve its underlying design and style while preserving the externally observable behaviour. "Code smells" help to detect inadequate structures and designs, which are then gradually removed through refactoring processes.

There is a prospect of aspect-oriented programming becoming a mainstream technology in the near future. This begs the question of how to deal with a large base of object-oriented legacy code when aspect-orientation becomes standard practice. AspectJ's backward compatibility with Java opens the way for refactoring existing Java applications to leverage the concepts and mechanisms of aspects. This requires a prior idea of good style for aspect-oriented source code, something yet to be developed and matured.

This thesis contributes to the definition of a new style appropriate to aspect orientation. To this effect, this thesis documents a collection of novel refactorings enabling the extraction of crosscutting concerns from object-oriented legacy source code and the subsequent restructuring of the aspects thus obtained. In addition, this thesis presents a review of traditional object-oriented code smells so they can be used as indicators of latent aspects in object-oriented source code. Finally, this thesis proposes several novel aspect-oriented code smells. We validate the refactorings through an illustrative refactoring process.

Keywords: Aspect-Oriented Programming, Refactoring, Object-Oriented Programming.

RESUMO

Idealmente, as ferramentas de engenharia de programas suportariam uma estrita separação de facetas, possibilitando a colocação de cada faceta na sua própria unidade modular. Infelizmente, as actuais ferramentas e linguagens – incluindo as que suportam o paradigma da orientação ao objecto – não conseguem obter uma completa e efectiva separação de todas as facetas. Daí resultam fenómenos indesejáveis tais como a dispersão e emaranhado de texto fonte.

A programação orientada ao aspecto é um novo paradigma da programação capaz de modularizar facetas transversais. A orientação ao aspecto complementa os paradigmas existentes, incluindo a orientação ao objecto, com mecanismos que providenciam uma separação de facetas mais completa.

A refabricação de programas é uma técnica para reestruturar o texto fonte de um programa no sentido de melhorar a concepção e estilo subjacentes, mantendo o seu comportamento externamente observável. "Maus cheiros" no texto fonte ajudam a detectar estruturas e concepções inadequadas, que são então gradualmente removidos através de processos de refabricação.

Existe a perspectiva da orientação ao aspecto ter uma aceitação generalizada no futuro próximo. Coloca-se a questão de como lidar com uma grande base instalada de texto fonte orientado ao objecto legado quando tal acontecer. A compatibilidade retroactiva de AspectJ em relação a Java possibilita a refabricação das aplicações Java existentes de modo a tomarem partido dos conceitos e mecanismos dos aspectos. Porém, isto tem como pressuposto uma ideia clara de bom estilo para o texto fonte orientado ao aspecto, algo que actualmente não existe duma maneira desenvolvida e matura.

Esta tese contribui para a caracterização de um novo estilo, apropriado à orientação ao aspecto. Para esse efeito, esta tese documenta uma colecção de refabricações originais através das quais facetas transversais existentes em texto legado orientado ao objecto são extraídas para aspectos, e a posterior reestruturação dos aspectos assim obtidos pode ser realizada. Esta tese apresenta também uma reapreciação dos maus cheiros orientados ao objecto tradicionais no sentido de poderem ser usados na detecção de aspectos latentes no texto fonte orientado ao objecto. Por fim, esta tese propõe diversos maus cheiros originais, orientados ao aspecto. As refabricações são validadas por meio de um processo de refabricação ilustrativo.

Palavras chave: Programação Orientada ao Aspecto, Refabricação, Programação Orientada ao Objecto.

ACKNOWLEDGMENTS

It was remarked before that there is nothing more rewarding than writing the acknowledgements section of a doctoral dissertation. Indeed, now that I'm writing this last section of my work, I can finally understand the joy!

First and foremost, I thank João Miguel Fernandes. More than a supervisor, he was my partner in a team that worked extremely well throughout the 37 months of this extraordinary journey. João Miguel kept bringing new challenges that I would not have dared attempting to tackle if it was not for his prompting, and which I, often to my surprise, kept meeting. He led me to fulfil a potential of which I was previously unaware. Without him, work for this thesis could never have evolved the way it has.

I'm deeply indebted to my parents, Maria Ângela and Jorge Pessoa Monteiro, for providing the office space near my home where I had the ideal conditions to work and think (including, thank God, isolation from the Internet!).

António M. F. Rito da Silva deserves a special place in this acknowledgements section. As leader of the Software Engineering Group at INESC-ID, he was extremely open and supportive since I approached him at the start of my work. Rito was instrumental in the definition of my work: his was the idea of putting together the subjects of aspect-oriented programming and refactoring.

My thanks to Sérgio Miguel Fernandes, for his patience and unfailing support while I was exploring the WorkSCo framework.

Stefan Hanenberg gave me a precious hint at the AOSD 2003 in Boston, by remarking that at the time nobody really had a clear idea of what a "good style" for AOP should be. His remark made me realise that I should tackle the subject of refactoring from the side of style.

Special thanks to João Luís Ferreira Sobral and Rui Mendes for their support and friendship.

This project was supported by PRODEP III (Medida 5 – Acção 5.3 – Eixo 3 – Formação Avançada de Docentes do Ensino Superior). I also acknowledge the financial support given by project PPC-VM (PO-SI/CHS/47158/ 2002) and by FLAD (Fundação Luso Americana para o Desenvolvimento).

Finally, I dedicate this thesis to my beloved wife Ana Maria and our children Maria Adília and Jorge Fernando.

CONTENTS

CHAPTER 1. INTRODUCTION **1**

1.1. MOTIVATION 1
1.2. PROBLEM STATEMENT 5
1.3. AIMS OF THE THESIS 7
1.4. APPROACH TAKEN 7
1.4.1 First Case Study: the WorkSCo Framework *7*
1.4.2 Second Case Study: the GoF Patterns *8*
1.5. CONTRIBUTIONS OF THE THESIS 8
1.6. ISSUES NOT ADDRESSED 9
1.7. ORGANISATION OF THE THESIS 9

CHAPTER 2. SEPARATION OF CONCERNS AND REFACTORING **11**

2.1. TRADITIONAL SEPARATION OF CONCERNS 11
2.1.1 Object-Oriented Programming *11*
2.1.2 The Law of Demeter *15*
2.1.3 Design Patterns *15*
2.1.4 Components *17*
2.1.5 Frameworks *20*
2.2. REFACTORING 20
2.2.1 Role of Units Tests in Refactoring *22*
2.2.2 Brief History of Refactoring *23*
2.2.3 Bad Smells in the Code *24*
2.2.4 Why do we need Refactoring? *26*
2.3. METAPROGRAMMING AND REFLECTIVE TECHNIQUES 27
2.4. ASPECT-ORIENTED PROGRAMMING 29
2.4.1 What is AOP? *30*
2.4.2 Concepts and Mechanisms of an Aspect-Oriented Language *32*
2.4.3 A Complete Illustrating Example *38*
2.4.4 Aspect Interfaces and Reusable Aspects *41*

CHAPTER 3. CASE STUDY: WORKSCO **45**

3.1. WHAT IS WORKSCO? 46
3.2. WHY WORKSCO WAS SELECTED 46
3.3. WHY WORK ON WORKSCO WAS DISCONTINUED 46
3.4. APPROACH TAKEN 47
3.5. ARCHITECTURE OF WORKSCO 47
3.6. EXPLORATION OF WORKSCO 49
3.6.1 The Data Link Feature *50*
3.6.2 Initial Hurdles *51*
3.6.3 Style Issues *53*
3.7. WORK ON THE CASE STUDY 54
3.7.1 Extracting the Data Link Feature *54*
3.7.2 Analysis of the Debugging Concern *60*
3.8. DISCUSSION 60
3.8.1 Good Object-Oriented Style is a Precondition for applying AOP *61*
3.8.2 Preservation of Intent vs. Preservation of Behaviour *61*

3.8.3 Feature Decomposition as a Likely Anti-Pattern *62*

CHAPTER 4. CASE STUDY: IMPLEMENTATIONS OF THE GOF PATTERNS IN JAVA AND ASPECTJ **65**

4.1. WHY THE EXAMPLES WERE SELECTED 65
4.2. APPROACH TAKEN 65
4.3. AD HOC ANALYSIS OF THE ASPECTJ IMPLEMENTATIONS 66
4.3.1 Role Assignment and Role Separation *66*
4.3.2 A Price for Reusability *69*
4.3.3 Difficulties in Capturing Context *70*
4.3.4 Problems of Privileged Access *74*
4.3.5 Problems with Illegal Weaving *74*
4.4. RESULTS DERIVED FROM THE CODE EXAMPLES 75
4.5. SUMMARY 75

CHAPTER 5. ASPECT-ORIENTED CODE SMELLS **77**

5.1. OO SMELLS IN THE LIGHT OF AOP 77
5.2. THE DOUBLE PERSONALITY CODE SMELL 78
5.3. ABSTRACT CLASSES AS A CODE SMELL 80
5.4. THE *ASPECT LAZINESS* CODE SMELL 80

CHAPTER 6. ASPECT-ORIENTED REFACTORINGS **85**

6.1. FORMAT OF THE REFACTORINGS 85
6.2. GENERAL GUIDELINES 86
6.3. REFACTORINGS FOR FEATURE EXTRACTION 87
6.3.1 Change Abstract Class to Interface *88*
6.3.2 Extract Feature into Aspect *90*
6.3.3 Extract Fragment into Advice *94*
6.3.4 Extract Inner Class to Standalone *99*
6.3.5 Inline Class within Aspect *102*
6.3.6 Inline Interface within Aspect *103*
6.3.7 Move Field from Class to Inter-type *104*
6.3.8 Move Method from Class to Inter-type *106*
6.3.9 Replace Implements with Declare Parents *108*
6.3.10 Split Abstract Class into Aspect and Interface *109*
6.4. REFACTORINGS FOR RESTRUCTURING THE INTERNALS OF ASPECTS 112
6.4.1 Extend Marker Interface with Signature *113*
6.4.2 Generalise Target Type with Marker Interface *114*
6.4.3 Introduce Aspect Protection *116*
6.4.4 Replace Inter-type Field with Aspect Map *118*
6.4.5 Replace Inter-type Method with Aspect Method *125*
6.4.6 Tidy Up Internal Aspect Structure *128*
6.5. REFACTORINGS TO DEAL WITH ASPECT GENERALISATION 129
6.5.1 Extract Superaspect *129*
6.5.2 Pull Up Advice *130*
6.5.3 Pull Up Declare Parents *130*
6.5.4 Pull Up Inter-type Declaration *131*
6.5.5 Pull Up Marker Interface *132*
6.5.6 Pull Up Pointcut *133*
6.5.7 Push Down Advice *133*
6.5.8 Push Down Declare Parents *134*

6.5.9 Push Down Inter-type Declaration *135*
6.5.10 Push Down Marker Interface *135*
6.5.11 Push Down Pointcut *136*
6.6. DEALING WITH LEGACY CODE 137
6.6.1 Partition Constructor Signature *137*

CHAPTER 7. VALIDATING EXAMPLE: REFACTORING OBSERVER 139

7.1. THE OBSERVER DESIGN PATTERN 139
7.2. THE FLOWER EXAMPLE 140
7.3. THE JAVA STANDARD API OBSERVER-OBSERVABLE PROTOCOL 141
7.4. THE ORIGINAL JAVA IMPLEMENTATION 142
7.5. UNIT-TESTING THE FLOWER EXAMPLE 145
7.6. THE REUSABLE AOP IMPLEMENTATION OF OBSERVER 149
7.7. REFACTORING SESSIONS 151
7.7.1 First Phase: Feature Extraction *151*
7.7.2 Second Phase: Restructure the Internals of the Aspect *160*
7.7.3 Third Phase: Extracting Common Code to a Superaspect *169*
7.7.4 Alternative Refactoring Path *171*
7.8. DISCUSSION 181

CHAPTER 8. RELATED WORK 183

8.1. GENERAL OVERVIEW 183
8.2. ASSESSING THE GOF PATTERN IMPLEMENTATIONS CODE 183
8.3. GUIDELINES ON HOW TO APPROACH THE SOURCE CODE 184
8.4. GENERAL STYLE RULES 184
8.5. PROPOSALS FOR ASPECT-ORIENTED REFACTORINGS 185
8.6. PROPOSALS FOR ASPECT-ORIENTED CODE SMELLS 186
8.7. PROVING THE PRESERVATION OF BEHAVIOUR 187
8.8. POTENTIAL PITFALLS 187

CHAPTER 9. CONCLUSIONS 189

9.1. CONTRIBUTIONS 189
9.2. PUBLICATIONS 190
9.3. FUTURE RESEARCH 190
9.3.1 Maturing the Refactorings *190*
9.3.2 Pinpointing Other Code Smells *191*
9.3.3 Developing Other Refactoring Ideas *191*
9.3.4 Covering Other Language Characteristics *192*
9.3.5 Refactorings for Restructuring the Remaining Base Code *192*
9.3.6 Refactorings to Cope with Published Interfaces *192*
9.3.7 Developing Opposite Refactorings *192*
9.3.8 Updating to New Language Versions *193*
9.3.9 Comparing AspectJ with AHEAD *193*

APPENDIX A REFACTORINGS DOCUMENTED BY FOWLER 195

APPENDIX B CODE SMELLS DESCRIBED BY BECK AND FOWLER 197

APPENDIX C ADDITIONAL REFACTORINGS FROM WWW.REFACTORING.COM 199

APPENDIX D REFACTORINGS TO PATTERNS BY KERIEVSKY 201

APPENDIX E CODE SMELLS DESCRIBED BY KERIEVSKY 203
APPENDIX F CODE SMELLS AND SYMPTOMS DESCRIBED BY WAKE .. 204
APPENDIX G – ASPECTJ QUICK REFERENCE 206
BIBLIOGRAPHY.. 209
REFERENCES FROM THE INTERNET... 221

CODE LISTINGS

LISTING 1: SIMPLE CLASS DEFINING A SIMPLE METHOD 28
LISTING 2: STRAIGHTFORWARD WAY TO CALL A METHOD 29
LISTING 3: CALLING A METHOD THROUGH REFLECTION 29
LISTING 4: SIMPLE EXAMPLE OF POINTCUTS AND ADVICE 36
LISTING 5: MESSAGES TO THE CONSOLE AS A CROSSCUTTING CONCERN 38
LISTING 6: EXAMPLE OF A SIMPLE ASPECT ENCLOSING MESSAGES IN BRACKETS 39
LISTING 7: EXAMPLE OF AN ASPECT SORTING MESSAGES TO THE CONSOLE 40
LISTING 8: REUSABLE ASPECTJ IMPLEMENTATION OF CHAIN OF RESPONSIBILITY 42
LISTING 9: SNAPSHOT OF THE COMPILE METHOD FOR THE FORKPROCEDURE CLASS. 52
LISTING 10: SNAPSHOT OF THE REPEATUNTILPROCEDURE CLASS' CONSTRUCTOR. 53
LISTING 11: SNAPSHOT OF A METHOD BETRAYING CONDITIONAL COMPLEXITY. 54
LISTING 12: VARIANT OF INTRODUCED CONSTRUCTOR WITH CALL TO SUPER. 56
LISTING 13: COMPOSITEPROTOCOL – REUSABLE IMPLEMENTATION OF COMPOSITE. 71
LISTING 14: COMMANDPROTOCOL – REUSABLE IMPLEMENTATION OF COMMAND. 72
LISTING 15: STANDALONE INTERFACES AUXILIARY TO COMMANDPROTOCOL. 73
LISTING 16: ILLUSTRATING USE CASE FOR THE COMMAND PATTERN. 73
LISTING 17: EXAMPLE OF THE *DOUBLE PERSONALITY* SMELL 79
LISTING 18: CODE EXAMPLE ILLUSTRATING THE *ASPECT LAZINESS* SMELL. 82
LISTING 19: CODE EXAMPLE AFTER REMOVING THE *ASPECT LAZINESS* SMELL. 83
LISTING 20: INITIAL ILLUSTRATING EXAMPLE FOR *EXTRACT FEATURE INTO ASPECT*. 92
LISTING 21: CLEAN CODE AFTER USING *EXTRACT FEATURE INTO ASPECT*. 93
LISTING 22: WINDOW VIEW CONCERN AFTER EXTRATION INTO AN ASPECT. 94
LISTING 23: PRECONDITION CHECKING CONCERN AFTER EXTRACTION TO AN ASPECT. 94
LISTING 24: INITIAL (TANGLED) FORM OF THE FLOWER SUBJECT CLASS. 143
LISTING 25: INITIAL (TANGLED) FORM OF THE BEE OBSERVER CLASS. 144
LISTING 26: INITIAL (TANGLED) VERSION OF THE HUMMINGBIRD OBSERVER CLASS 144
LISTING 27: TEST METHOD USED THROUGHOUT AS CLIENT CODE. 145
LISTING 28: OUTPUTSUPRESS ABSTRACT ASPECT 146
LISTING 29: OUTPUTCAPTURE ABSTRACT ASPECT 147
LISTING 30: POINTCUT ASPECT OF GENERALLY APPLICABLE POINTCUTS 147
LISTING 31: TEST CLASS FOR THE FLOWER EXAMPLE OF THE OBSERVER PATTERN 148
LISTING 32: THE REUSABLE OBSERVERPROTOCOL ASBTRACT ASPECT 150
LISTING 33: ONE CONCRETE SUBASPECT OF OBSERVERPROTOCOL 150
LISTING 34: CLEAN VERSION OF THE FLOWER CLASS. 156
LISTING 35: CLEAN VERSION OF THE BEE CLASS 157
LISTING 36: CLEAN VERSION OF THE HUMMINGBIRD CLASS 158
LISTING 37: OBSERVINGOPEN JUST AFTER THE EXTRACTION PROCESS 159
LISTING 38: OBSERVINGOPEN ASPECT AFTER USING *GENERALISE TARGET TYPE WITH MARKER INTERFACE* 163
LISTING 39: OBSERVINGOPEN AFTER BEING COMPLETELY TIDIED UP 166
LISTING 40: OBSERVINGCLOSE AFTER BEING COMPLETELY TIDIED UP 168
LISTING 41: OBSERVINGRELATIONSHIPS ABSTRACT ASPECT 170
LISTING 42: FINAL FORM OF THE OBSERVINGOPEN ASPECT 170
LISTING 43: FINAL FORM OF THE OBSERVINGCLOSE ASPECT 171
LISTING 44: OBSERVINGOPEN AS SUBASPECT OF OBSERVERPROTOCOL 173
LISTING 45: OBSERVINGOPEN AFTER SOME TIDYING UP 176

LISTING 46: RESTRUCTURED VERSION OF TEST CLASS. 178
LISTING 47: NEW TEST FOR THE SECOND IMPLEMENTATION OF OBSERVINGOPEN......... 179
LISTING 48: SECOND FINAL VERSION OF OBSERVINGOPEN. 180
LISTING 49: FINAL VERSION OF OBSERVINGCLOSE 180
LISTING 50: REFACTORED VERSION OF THE TEST FOR THE FLOWER EXAMPLE............... 181

FIGURES

Figure 1: Example of good modularity – XML parsing in org.apache.tomcat. . 4
Figure 2: Example of crosscutting – Logging in org.apache.tomcat. 4
Figure 3: JUnit's red bar signalling a failed test.. 23
Figure 4: Introdutory Example for the *Compose Method* refactoring. 25
Figure 5: (a) Traditional Compilation (b) Compilation with Weaving............. 31
Figure 6: General structure of reusable aspects 43
Figure 7: The WorkSCo front-end procedure hierarchy. 48
Figure 8: Architecture of the WorkSCo framework 49
Figure 9: Original chain of procedure constructors in WorkSCo. 55
Figure 10: Constructor Hierarchies with and without Data Links 56
Figure 11: Dealing with a crosscutting concern internal to an aspect. 57
Figure 12: Structure for the Observer Design Pattern. 140

TABLES

TABLE 1: EXAMPLES OF POINTCUTS IN ASPECTJ. 34
TABLE 2: EXAMPLES OF CONTEXT CAPTURE BY POINTCUTS. 35
TABLE 3: EXAMPLES OF CONTEXT CAPTURE BY POINTCUTS. 35
TABLE 4: ANALYSIS OF ASPECTJ IMPLEMENTATIONS OF THE GOF PATTERNS ACCORDING TO ROLES AND MODULARITY PROPERTIES 68

LIST OF ABBREVIATIONS

AHEAD	–	Algebraic Hierarchical Equations for Applications Design
AO	–	Aspect-Oriented
AOP	–	Aspect-Oriented Programming
AOSD	–	Aspect-Oriented Software Development
API	–	Application Program Interface
ASoC	–	Advanced Separation of Concerns
DSL	–	Domain-Specific Language
GoF	–	Gang-of-Four (the four authors of [48])
IC	–	Integrated Circuit
IDE	–	Integrated Development Environment
OO	–	Object-Oriented
OOP	–	Object-Oriented Programming
XML	–	Extended Mark-up Language
XP	–	Extreme Programming
WfDL	–	Workflow definition language
WorkSCo	–	Workflow for separation of concerns

Chapter 1. Introduction

Aspect-Oriented Programming (AOP) [86] is a new paradigm that is presently gaining increasing acceptance in the software engineering community. There is the prospect of widespread adoption of aspect-orientation in the not too distant future, which begs the question of how to deal with a large legacy of object-oriented (OO) code. Experience with refactoring of OO software in the last half-decade suggests that refactoring techniques have the potential to bring the concepts and mechanisms of aspect-orientation to existing OO frameworks and applications. However, an essential prerequisite for the use of refactoring is presently missing: a clear notion of "good" aspect-oriented style, as opposed to existing notions of OO style. Such a style would be useful both to refactor existing OO code to turn it into aspect-oriented code, and to improve aspect-oriented code.

In recent years, a clear notion of good style for OO code bases was successfully represented through catalogues of code smells [47] – symptoms in the source code that can be used to detect the existence of potential problems – and refactorings [125][47] – behaviour-preserving transformations of the source code that remove the smells, thus improving the underlying design.

This dissertation derives collections of refactorings and code smells that contribute to characterise a style specific to aspect orientation.

1.1. Motivation

Ever since the beginning of software engineering as an independent activity, one of its main goals has been to bridge the gap between computer hardware and the human mind. The programming models developed in recent decades show a steady path from the low level of computer hardware towards the way the human mind functions. All efforts to make programming easier are ultimately efforts to find a model suitable for the human mind, and translate instances of such a model to the low-level imperative code that machines understand. From the basis of this fact, some authors are studying the referencing and binding mechanisms of natural languages to derive useful insights for programming language design [101].

How does the human mind work? It is important to consider its limitations as well as its strengths. The human mind clearly has limits regarding the quantity of details it can cope with at a given moment. This explains why models are needed. Models enable us to concentrate on the relevant features of the problem in hand, without having to consider other characteristics. In a word, our minds need *abstraction*.

As Kiczales remarked in [81], abstraction is not only a primary concept in all engineering disciplines, but also a basic property of how people approach the world. Kiczales noted that, in software, abstractions could be used as (a) a tool for managing complexity, (b) a convention to draw clear boundaries between the different aspects (concerns) of a system, (c) the appropriate way for provider programmers to give component software to client programmers, without overburdening them with "unnecessary" details.

It is no coincidence that one persistent aim of software engineering throughout its development has been the creation and perfection of models and tools for the support of abstraction. We cannot concentrate on many subjects or concerns simultaneously. Instead, we need to *abstract* from most of the concerns so that we can concentrate on a particular concern in order to deal with it effectively. We need to break, or decompose systems and

problems into smaller subsystems and sub-problems, in order to concentrate on each sub problem by turns. This is the fundamental idea behind the works of Parnas [132] and Dijkstra [37] when they suggested *separation of concerns*[1] as a technique for problem solving. They were simply echoing the way the human mind functions.

A concern is any issue from a system's design potentially deserving of the attention of the programmer at a given time during the design, implementation and maintenance phases. Separation of concerns is the ability to keep every concern in its own *unit of modularity*, i.e. constructs in a programming language containing code that can potentially be part of one or more programs, have a well-defined interface and are amenable to separate treatment by available tools. To *modularise* a concern is to enclose all code related to that concern within a unit of modularity for the sake of its own consistency and to ease the human programmer's task of reasoning with it. Modularity greatly facilitates system development because if we are able to reason with an issue of the system at a time, while keeping in mind that we are dealing with only one of such issues, we are making things easier for ourselves.

Ideally, the decomposition of a system into separate concerns would lead directly to an optimal structure of the intended system. In practice, however, it will be possible to implement the resulting structure exactly as designed only if the supporting tools provide the necessary composition mechanisms. Those mechanisms determine what we can separate, because we will not separate what we cannot later compose. In this context, we use the term *composition* to refer to the ability to bind or put together software parts (i.e. modules) that were separately developed. Examples of language-level composition of software parts include widely known mechanisms such as method or procedure calls and class inheritance.

The fact that a given language does not provide a direct support for a particular composition does not necessarily mean that the intended effect cannot be achieved from the mechanisms available with that language. However, in such cases the intended effects are achieved only through redundancy and/or additional efforts on the part of the programmer. For instance, one of the crudest forms to achieve a composition is through code duplication, i.e. copying and pasting the section of code that performs the desired sequence of actions in all the points in the program's code where we want such actions to take place. Naturally, such a practice goes against all modern notions of good programming practice, but it is worth noting that we can afford to make that judgement only because we presently have more powerful composition mechanisms at our disposal. In this case, we would place the relevant code in a routine or method, and then place calls to that routine or method in all relevant places.

A well-known example of a language-level composition mechanism replacing redundant code is dynamic binding in object-oriented languages (see section 2.1.1). It is possible to emulate its effect in procedural languages such as Pascal, by using conditional control structures (typically case statements or chains of if-then-else structures). However, conditional statements are a much less satisfactory way because any addition to, or change in, the available cases entails *invasive* modifications to the source code of those statements. Software developers strive for composition mechanisms that achieve their intended effects in a modular way, i.e. without requiring invasive changes – as is the case of dynamic binding.

[1] Parnas is generally credited for the introduction of the concept of separation of concerns, when proposing modular programming [132] as a better way to structure program code. The term "separation of concerns" was first coined by Dijkstra in [37].

In order to achieve compositions in a modular way, the composition mechanisms need units of modularity. Functions, procedures, modules, methods, classes and packages are examples of units of modularity. Available composition mechanisms work on the available units of modularity, and therefore it is important to consider what code can and cannot be placed within such units.

Unfortunately, current software development tools, including OO languages, do not provide composition mechanisms that are powerful enough to enable the desired level of separation of concerns. Code duplication is a phenomenon that still occurs in a significant – and perhaps increasing – number of occasions in present software engineering activities. The example of methods and method calls comprises one suitable example – although in object-oriented languages we can avoid duplicating the code of the method, we still have to duplicate the calls to the method in an unbounded number of places.

When not all concerns of a system's code can be kept separate, several negative properties ensue. Code related to a concern that cannot be modularised usually results in *code scattering* – small code fragments scattered throughout several units of modularity. Concerns that suffer from this problem are known in the literature as *crosscutting concerns* [86], as its code cuts across several units of modularity. Furthermore, the available units will enclose code relative to several other concerns, resulting in an intertwined mixture of code fragments usually referred as *code tangling*. Scattering and tangling both increase the difficulty in reading, understanding, analysing, maintaining, and reusing code. These limitations are the root cause of problems such as the so-called *inheritance anomalies* [106][3], which started to be noticed at the end of the 1980s. Inheritance anomalies motivated an enormous quantity of research efforts and publications during the 1990s, e.g. [108][18][144][66].

Figure 1 shows an example of good modularity. It represents the code of one of the versions of org.apache.tomcat server[2]. The vertical bars depict individual modules and the red lines depict the lines of code related to extended mark-up language (XML) parsing. We can see that all the code related to the concern is placed inside a single module, which contains only code related to that concern. Whenever we need to change something related to this concern, we know that all changes should be done within that module. This contrasts with another concern found in the same code base, shown in Figure 2. The red lines depict the code related to logging, which is spread throughout the entire system. Hence, there is duplication of code to address the same concern at a number of non-contiguous points within the system. Logging is a crosscutting concern.

It is highly desirable that all the relevant concerns are considered and planned for as early as possible. In this way, it is easier to conceive an architecture that takes into account all concerns, maximising flexibility and maintainability. However, experience showed that it is not generally possible to foresee all concerns upfront in the design phase. Some concerns are revealed or identified only after experience is gained during implementation. When some late concern appears, its code has to be manually inserted into the already existing code. As a result, the underlying design suffers in many occasions, and the scattering and tangling problems increase.

To compound the problem, requirements tend to change over time [133][56], sometimes even during the analysis or design phases. This means that even when all relevant concerns are identified – and therefore duly taken into account in the software architecture – a changing requirement may cause the same problems as if a more careless design had been produced [56].

[2] Figure 1 and Figure 2 were taken from a presentation by the Xerox PARC team that developed the early versions of AspectJ.

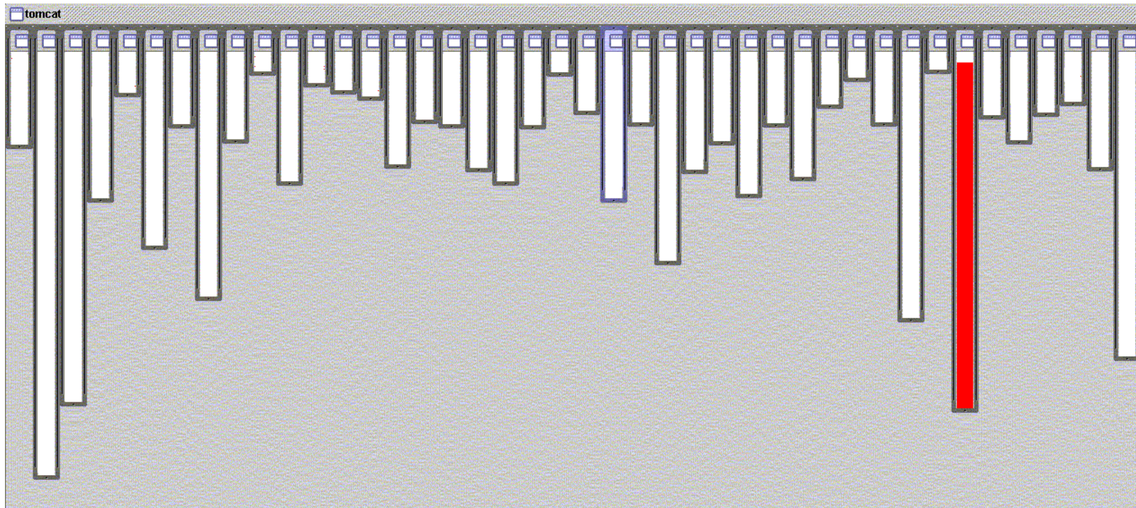

Figure 1: Example of good modularity – XML parsing in org.apache.tomcat.

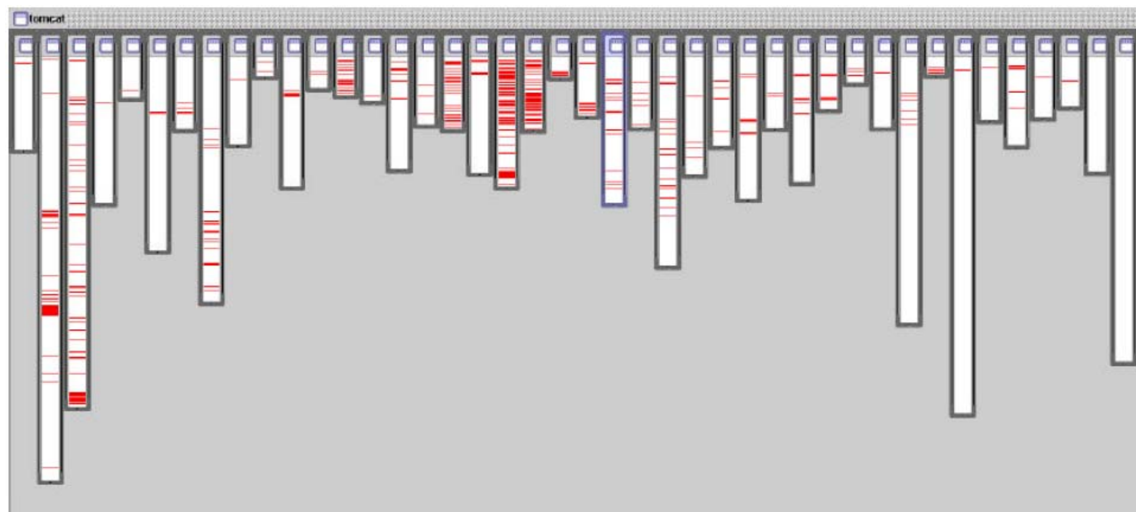

Figure 2: Example of crosscutting – Logging in org.apache.tomcat.

To further compound the problem, sometimes a given functionality may be needed in various contexts, which may require different sets of additional concerns (e.g. different error-handling strategies, synchronisation policies, distribution strategies in network environments, different supports for monitoring, logging, debugging, optimisation, etc), each with its own specific additional code. In such cases, it may not always be possible to reuse the code for the basic functionality in the various contexts [34]. Each may require a different version, with its own set of specific add-ons to the code. Even when a common system for all the required combinations can be built, it is likely to be more complex than would be strictly necessary for each individual combination, its design including a number of "hooks" and *hot spots* [135] – structural elements enabling software engineers to plug additional components or provide alternative implementations of specific features. These elements increase the complexity of the system's structure and are sometimes a cause for the deterioration of its performance.

All the aforementioned problems help to explain why the tangling effect tends to occur in later phases of the system development [34]. The design usually starts simple and clean, but as additional concerns are added into the system, both the evolving code and the design (when design documentation is still kept at all) gradually become more and more tangled and confused. Thus, program code tends to loose intentionality as it evolves.

The term *intentionality* refers to the conceptual gap between program code and the domain concepts of the design. The more intentionality we keep, the easier the program code is to be understood, modified and reused. Maximum intentionality can be attained through domain specific languages (DSLs), i.e. languages specifically designed to provide direct support to the concepts of a particular problem domain [34].

When we consider general-purpose programming languages such as C++ [147], CLOS [75] or Smalltalk [50], some loss of intentionality is bound to occur, since the programmer has to translate the concepts of the problem domain into the instructions and constructs supported by the language. Beyond that, however, the conceptual gap could theoretically be kept at a minimum, provided the code related to each concept is kept separate from the code related to other concepts. However, that cannot be attained with the technologies currently used in industrial and enterprise environments. For that, we need more powerful composition mechanisms. AOP is a new technique that improves on object-oriented programming (OOP) in this regard.

Intentionality is likewise a central concept in refactoring. Refactoring is one of the key concepts of *Extreme Programming* (XP) [12], which regards a system's source code as primarily a communication mechanism between humans rather than machines. Refactoring is a code-centric approach that aims to maximise the clarity of the expression of intent in source code [4]. Again, the emphasis is on making things easier for the human mind.

1.2. Problem Statement

This thesis explores the combination of AOP [40][86] and refactoring [184][47][53][125], two techniques that contribute to deal with the problems of permanent evolution of software. Refactoring is an approach that facilitates the continuous change of source code, enabling it to evolve in line with changes in environments and requirements. AOP provides stronger modularisation and software composition mechanisms than those provided by previous technologies, thus diminishing the potential impact that changes to the code related to a given concern have on code not related to that concern.

AOP's steady progress from "bleeding edge" research field to mainstream technology [140] brings forward the problem of how to deal with large number of OO legacy code bases. Experience with refactoring of OO software in the last half-decade suggests that refactoring techniques have the potential to bring the concepts and mechanisms of aspect-orientation to existing OO frameworks and applications. However, several interrelated hurdles will have to be overcome before developers of aspect-oriented software can employ refactoring in an effective way.

One hurdle is the present lack of a fully developed idea of "good" style for AOP. In practice, this entails developing a notion of good style for the AspectJ programming language [83][82][172]. AspectJ is an extension of the Java language [51] that includes mechanisms to support the concepts of AOP and is the most mature aspect-oriented language currently available. AspectJ therefore comprises the most obvious subject for the study of an aspect-oriented (AO) style. Throughout this thesis, AspectJ is used and regarded as the primary representative of AOP. This is reinforced by the fact that other AO tools tend to follow the example of AspectJ and support similar concepts [77]. Naturally, equating AspectJ with AOP is a simplification: future developments are likely to yield a greater variety of mature AO tools. When that happens, AspectJ will not dominate the AO landscape to the extent that it does today. However, we saw feet to use the terms AOP and AspectJ interchangeably in order to simplify and facilitate the presentation of the various concepts.

The ideas of refactoring and coding style are intimately connected. To a large extent, refactoring is transforming code written in bad style into code written in good style. Style rules [156] say how a human programmer should shape her source code so that fellow programmers do not have difficulty in understanding it, whenever they need to use it or modify it. We cannot rely on the compiler for this, as the rules enforced by the compiler merely tell us what machines understand. Style rules tells us what humans understand. In addition, style rules guide the programmer through the refactoring processes. Programmers won't be able to carry out a refactoring process unless they have a clear idea of what they are aiming at.

Why does AOP need its own specific style, different from that of OOP? Coding styles aim to express the coding practices that yield source code easier to maintain and evolve. Whenever a programming language provides alternative ways to achieve some result, the way that causes the least problems to present and future programmers should be considered the one in best style. Throughout the various stages of development of programming languages, many developments of style were motivated by the advent of new, superior mechanisms. We briefly mention three examples:

1. Dijkstra's famous dictum that the "Go-to statement [should be] considered harmful" [38] stemmed from the availability of control structures namely loops.
2. Fowler et al. [47] considered the use of the switch statement to be a code smell, due to the availability of polymorphism and dynamic binding.
3. Orleans suggested in [128] that the 'if' statement be considered harmful in the context of languages using elaborate forms of predicate dispatch.

All these considerations suggest that the appropriate notion of style for a given language strongly depends on what can be achieved with that language. In this light, the suitable style of AspectJ cannot be the same as for Java. AspectJ enables programmers to perform compositions that are impossible with Java, most notably the modularisation of crosscutting concerns. This modularisation eliminates negative qualities that result from the crosscutting effect, such as code scattering and code tangling, which we should not expect to find in aspect-oriented source code. This suggests that many of traditional OO solutions, which betray such negative qualities, should now be considered bad style.

The absence of a notion of style for AO programs is a consequence of the fact that we still do have a complete and mature knowledge of the new paradigm. We still do not know how best to use each construct of AspectJ, how the various constructs interact and what are the implications of those interactions. We still do not have a clear knowledge of what the relationship between "base code" and aspects should be, and how best to leverage aspects in designing interfaces and building libraries and frameworks.

The very compositional power of AspectJ can be cause for problems. AspectJ offers multiple ways to achieve various effects and compositions. For instance, the implementation of mixins [22] can be achieved both through marker interfaces controlled within the aspect and through inner static aspects placed within interfaces. Likewise, non-singleton aspect associations provide alternatives to solutions obtained with the default singleton aspects. AspectJ programmers are sometimes faced with so many choices that it becomes hard to decide on the design most appropriate to a particular situation. There is a need to further study the consequences and implications of each solution in order to make choices clear. We believe that catalogues of code smells and refactorings are an effective way to present this knowledge to programmers.

For human programmers to have a clear notion of style, they need a consistent set of rules that effectively captures and expresses it. The way chosen by Fowler et al. to express the

style they advocate in [47] was through a catalogues of *code smells* (see Chapter 5) – symptoms in the source code serving as indicators that something may not be as it should be. These smells are compounded by a catalogue of refactorings (see Chapter 6) through which those smells can be removed from existing code. These catalogues proved very useful in bringing the concepts of refactoring and good OO style to a wider audience and in providing programmers with guidelines on when to refactor and how best to refactor. Presently, an AO equivalent of such catalogues is not available. This constitutes another hurdle for the development and adoption of Aspect-Oriented Refactoring and is both a cause and a consequence of the first.

1.3. Aims of the Thesis

The broad aim of this thesis is to expand the current refactoring space, which is currently in its infancy. The primary specific aim is to develop a collection of refactorings for aspect-oriented source code, using AspectJ as the subject programming language.

Transforming the source code in order to improve it assumes the existence of a clear idea of when code needs improving. In this thesis, we use the notion of *refactoring* as the transformation steps required to remove *bad smells* from existing source code. We therefore set the complementary aim of developing a set of code smells pinpointing the situations in source code that warrant the use of the refactorings.

1.4. Approach Taken

We took the approach of using case studies as a vehicle for gaining the necessary insights. The case studies should be code bases in AspectJ or Java with the appropriate structural characteristics (e.g. presence of crosscutting concerns). We approached the Java code bases as if they were AspectJ code bases written in a bad style (i.e. Java code as "smelly" AspectJ code). The case studies would be subject to refactoring experiments to derive the kinds of refactorings that would be effective in turn them into examples of good AO style (i.e. that would remove the code smells).

Some criteria were set for the selection of the case studies: all case studies should be independent of the author, i.e. created neither by him nor under his specifications. At least one of these must comprise a "real" example, being non-trivial in both size and complexity. At the start of our work [116], we selected two candidates to be used as case studies: (1) the WorkSCo framework [187] and (2) the collection of implementations (version 1.1) in both Java and AspectJ of the 23 Gang-of-Four (GoF) design patterns [48], presented by Hannemann and Kiczales [60].

1.4.1 First Case Study: the WorkSCo Framework

The WorkSCo framework is an instance of the architecture proposed in [104]. It is being developed at the ESW [186] group at INESC-ID, using Java technology. Workflow applications [65] are a good example of modern software, whose requirements include a multitude of crosscutting concerns. They stress to the limit the capabilities for separation of concerns of current software engineering technologies.

The architecture of WorkSCo aims to provide a flexible support for evolution. Its design relies on a considerable number of design patterns, most of which are intended to achieve particular separations of concerns. The framework as implemented at ESW/INESC-ID [186] introduces a few structural deviations from the original design [104][103] in order to achieve separations that are more ambitious [43]. WorkSCo can be regarded as an example

of the maximum results of separation of concerns that can be obtained using conventional approaches such as those surveyed in section 2.1 of this Chapter – including OOP, components and design patterns. That made WorkSCo an interesting case study as well as an appropriate starting point for the research presented in this thesis.

WorkSCo was used as the first case study. The most significant work on WorkSCo comprised the extraction of one crosscutting concern from the code base. The first refactorings documented in this thesis were derived from these extraction experiments.

1.4.2 Second Case Study: the GoF Patterns

Over the last decade, OO *design patterns* [48] became a popular approach to reusing design knowledge. Patterns are a means to capture the experience and knowledge of software designers. A pattern names and describes a problem commonly found in software designs, and prescribes a range of variants of a solution for the problem, so as to ease the reuse of that solution [135][23][143]. One of the variants of the solution is applied in each different design, adapted to the specifics to the given problem. The repetitive use of a pattern over many designs yields many design structures that have a set of core structural characteristics in common.

The idea behind design patterns is to record the essence of solutions to recurrent design problems in order to facilitate their reuse whenever one instance of the problem is encountered. Experienced designers captured successful design experience in pattern catalogues. By applying a solution that was successfully applied in the past, designers are free from thinking anew every time the problem recurs.

Several catalogues of patterns have been published, the most popular and influential of which is the book by Gamma et al. [48]. The popularity of this book and of the patterns it documents led many programmers across the world to develop their own implementations of the patterns in various languages, including Java [39][177][178][180][181][52]. In 2002, Hannemann and Kiczales presented their implementations in both Java and AspectJ and provided an analysis of the results [60]. Our work on the GoF patterns was centred on these dual implementations, compounded with other Java implementations [39][32], which were used to test and refine the results derived. A substantial part of the findings presented in this thesis originated from our study of these examples.

1.5. Contributions of the Thesis

The main results of this Ph.D. project are the following:

- A collection of twenty-eight refactorings for the AspectJ programming language.
- A review of the traditional OO code smells in light of AOP.
- The proposal of several novel code smells, including one that is specific to aspects.
- Several considerations associated with the above refactorings and code smells.

The collection of twenty-eight refactorings is structured in the following groups:

- Ten refactorings for the extraction of crosscutting concerns from Java code bases to aspects.
- Six refactorings for improving the internals of aspects, including aspects resulting from extraction processes performed according to refactorings of the previous group.

- Eleven refactorings dealing with the extraction of common code between multiple aspects and the associated transfer of aspect-specific constructs between superaspects and subaspects.
- One refactoring dealing with the separation of concerns in the signature of constructors that are part of published interfaces.

This thesis aims to contribute to the expansion of the aspect-oriented refactoring space, not to cover it completely or thoroughly (that was not yet achieved even for the Java language). We believe the development of a style for a new programming paradigm requires the accumulated experience of many researchers and practitioners across the software engineering community. The aim of this thesis is to provide a contribution.

1.6. Issues not addressed

The scope of work in the field of aspect-oriented refactoring is extensive and no thesis can attempt to deal with more than a subset of that field. It is important to understand what this thesis does not attempt to do.

- No aspect mining
  The term *aspect mining* is usually used to refer to the development of tools that perform an automatic or semi-automatic detection of crosscutting concerns [142][166][164] or assist the developer in such searches [138]. We do not contribute to the development of aspect mining tools, though we provide insights to the detection of latent aspects in the form of code smells, which are proposed in Chapter 4.
- No tool support
  Developing tools that automate standard transformations of source code is related to the one covered in this thesis, but is not the same. Even when provided with refactoring tools, developers still need to have a proper notion of style to decide when code transformations should be used, and to choose the specific transformation that must be applied to each specific circumstance. It is this knowledge that we aim to expand, from which tool developers can benefit. Though we developed the refactorings to be performed manually, we believe they can be helpful to developers of tool support for aspect-oriented refactorings [61][58][41][73], by suggesting refactorings that may be worthy of their development efforts. Therefore, this thesis indirectly contributes to the development of future tools.
- No metrics
  This thesis does not attempt to formally measure and quantify the benefits in code of the proposed refactorings. Work on metrics for the complexity of aspect-oriented source code can be found in [167], [165] and [49].
- No formalism
  This thesis does not attempt to provide a formal, mathematical basis for the refactorings. Cole and Borba are working in this field [30], and in [29] they state their intention of extending their work to cover our refactorings in future work.

With the above excluded the context of this thesis will hopefully be clearer.

1.7. Organisation of the Thesis

This thesis is organised as follows:

Chapter 2 starts with a brief overview of several traditional techniques and mechanisms used to achieve separation of concerns, with a stress on limitations and associated problems. Next, it presents refactoring as a technique to cope with some of those limitations. Reflection is briefly mentioned, since during a certain period it seemed to be a promising technique to overcome those limitations. Finally, we introduce aspect-oriented programming, which effectively realised the hopes initially placed on reflection. AspectJ is used as the primary representative of AOP.

Chapter 3 presents the first case study, the WorkSCo framework. The Chapter describes WorkSCo, focusing on the characteristics that motivated its use as a case study. We next explain why work on WorkSCo was discontinued, after which we describe how we approached WorkSCo. We describe its architecture, the experiments performed on it, hurdles encountered and results obtained. In the end, we present the conclusions we derived from the case study.

Chapter 4 presents the second case study, a collection of code examples comprising the implementation in Java and AspectJ of the GoF patterns. The Chapter explains why the examples were selected and how we approached them. We provide some background on the examples, and we describe various issues that we detected during our analysis and subsequent work. We briefly state the results derived and conclude the Chapter with a summary of the main insights we derived from the case study.

Chapter 5 presents some code smells that act as guidelines for using the refactorings. These include (1) traditional OO smells reviewed in the light of AOP, (2) novel smells designed to assist in the detection of crosscutting concerns in legacy OO code, and (3) an aspect-specific smell. The refactorings that can remove the smells are mentioned in the pertinent places.

Chapter 6 presents the refactorings. The Chapter presents the format used to describe the refactorings and presents various guidelines that are applicable to most refactoring processes. Next, the Chapter describes each of the refactorings, which are divided into four groups: (1) extraction of crosscutting concerns to aspects, (2) tidying up the internal structure of extracted aspects, (3) factoring out of common code to superaspects and (4) a special case dealing with constructors with published interfaces.

Chapter 7 presents a validating example that describes in detail a complete refactoring process targeting a small example based of the Observer pattern ([48], p. 293).

Chapter 8 surveys related work.

Chapter 9 concludes this thesis by summarising the contributions, mentioning the related articles so far published or submitted, and surveys opportunities for future work.

Chapter 2. Separation of Concerns and Refactoring

The purpose of this Chapter is to survey both traditional and novel techniques and mechanisms to achieve separation of concerns in software systems. Section 2.1 covers traditional techniques, such as OOP, design patterns, components and frameworks. Some explanations of why these techniques are not powerful enough are provided. Section 2.2 covers refactoring, a technique that helps to ameliorate the consequences of the limitations and shortcomings of the previously covered techniques. Reflection is mentioned in section 2.3, with a stress on the reason why many people regard it as a link between traditional techniques and AOP. Finally, AOP is described in section 2.4.

We do not claim that techniques such as AOP and refactoring solve or overcome all of the limitations and shortcomings mentioned in relation to the traditional techniques. Describing current problems and limitations serves to drive home the idea that a broad range of issues in software engineering are still in need of addressing. Nevertheless, we believe that techniques such as refactoring and AOP can ameliorate some of the problems and overcome some of the limitations.

2.1. Traditional Separation of Concerns

This section briefly surveys several traditional techniques of software engineering, with a stress on some of the limitations and associated problems. Subsections 2.1.1 covers object technology [68] and subsection 2.1.2 refers to the law of Demeter, which is closely related to OOP. Subsection 2.1.3 covers design patterns [48][135][23], subsection 2.1.4 discusses the concept of component [149][161] and subsection 2.1.5 briefly discusses frameworks [69][42].

2.1.1 Object-Oriented Programming

OOP is the current dominant programming paradigm[3], so much so that other software development techniques are often discussed in terms of the concepts of OOP, even though they are independent. Examples are design patterns, components, frameworks and refactoring.

Core Concepts

In the early years of research, there were uncertainties on exactly which features are essential to the definition of object-orientation. While a relative consensus was eventually reached regarding a number of "core" concepts [160], including object identity [79][80], modularity, implementation hiding, encapsulation, inheritance and polymorphism, the same did not happen to other features such as delegation [97][21], garbage collection[4], multiple inheritance, mixin inheritance [22], generic types [112] and design by contract [110][113].

[3] Currently, whenever a new programming language is created, it is expected to be object-oriented, unless there are specific reasons for not being so. This leads us to conclude that OOP is indeed dominant. The advent of AOP does not change that, because AOP complements existing paradigms rather than replace them.

[4] Garbage collection seems to be a later addition to the consensus, especially after the negative experiences with C++, the only major language supporting the concepts of object-orientation that does not provide some form of automatic memory management.

The latter features can be found in some OO languages, but not in others. Even concepts such as class, inheritance and polymorphism are not directly supported in all the models proposed through the years [97]. In consequence, OO languages vary more markedly than is the in previous, simpler, paradigms such as procedural programming.

One of the most fundamental concepts of OO is *object identity*. Each object has its own identity, which enables it to be distinguished from other objects, even when compared with an object with identical structure and holding similar state. Identity enables objects to be referenced by multiple other objects, and whenever an object changes its state, all other objects referencing it can know about the change, without confusing it for a different object. Associated to the concept of identity is the *self-variable* (variously called self, current or this), a variable through which the methods of the object can access the object's state and behaviour, without ever confusing it with the state and behaviour of a different object.

The term *class* has been used to represent several distinct concepts; depending on the author or language concerned. It is usual for authors and languages to assign more than one of these meanings to the term class, though not necessarily all of them. Thus, a class can be:

- A *type* specifying a set of objects that expose the same functionality and behaviour. It is usual to regard types as specifications of behaviour [113], independent of any particular implementation of that behaviour. One modern example of a construct modelling a type is the Java interface. However, Java interfaces are a purely syntactic mechanism, and lack a semantic dimension. A more complete mechanism to specify (and enforce) types can be found in the Eiffel language [111], which includes a system of operation pre-conditions and post-conditions, as well as class invariants. Together, these mechanisms provide a semantic dimension to type specification and enable design by contract.
- A *factory* or *template* [160][80] for the creation of objects of similar structure and behaviour, often called the *instances* of the class. Some OO languages enable classes to exist as independent entities during a program run, and some languages (e.g. Smalltalk) even enable classes to be changed in such a way that the next instances have different or additional properties [46].
- A *module* [145][113], comprising a repository of implementation (i.e. a set of variables and operations defined within a single lexical scope) that is hidden behind a well-defined interface. In this context, it is usual to view classes as entities that encapsulate a particular implementation of the behaviour specified by a type. In addition, classes can have one or more interfaces (i.e. classes can have more than one type). Entities from the environment outside of the module must interact with the instances solely through these interfaces. The modules can be self-contained parts of programs and can be used in different programs.

Meyer coined the open-closed principle[5] [113], which states that "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." This principle states that the behaviour of a module should be extensible, meaning that programmers should be able to make the module behave in new and different ways, as the requirements of the application change, or new requirements arise. In addition, the source code of the module should be inviolate, meaning no one should be allowed to make changes in its source code for the purposes of adding new functionality. It would seem that

[5] Meyer proposed the Open-Closed principle in the first edition of [113] (1988).

these two requirements are incompatible, but the mechanism of *inheritance* is capable of meeting both.

Inheritance is the mechanism through which a class shares its state and behaviour with another class – the former is variously called the superclass, parent or ancestor, and the latter is variously called the subclass, base class, child, heir or descendant. The fields (also variously called slots, attributes, member variables or instance variables) comprise the state and the methods (also called selectors, member functions or routines) implement the operations provided by the class, which together comprise the class' behaviour. By inheriting from the superclass, the subclass automatically acquires its fields and methods, without the need for the programmer to manually insert duplications of its definitions. Those definitions stay in just one place in the code, within the lexical scope of the superclass. OO languages usually allow the subclass to define new fields and methods, and to *override* the methods it inherits with implementations specific to the state and situations the subclass is intended to address. In most languages, overriding methods can also call the original method from the superclass (via a pseudo-variable usually called "super"), which enables programmers to extend inherited operations with new behaviour. It is considered good style to ensure that the overriding methods provide the same behaviour that would be expected of the superclass' method (i.e. the subclass honours the superclass' *contract*): The majority of OO languages does not provide mechanisms to automatically verify that contracts are enforced, beyond what is expressed through syntactic mechanisms. The major exception is Eiffel.

Dynamic binding automatically selects the appropriate section of code associated to a method, based of the runtime type of the object that was the target of the method call. Compilers cannot statically determine which function is called, because the information on the type of the object is available only during runtime. Therefore, some method lookup must be performed at runtime in order to dispatch the right method. The resulting effect is what is usually called dynamic or late binding. Though less flexible, static binding is usually a more efficient mechanism than dynamic binding, because all the compilers need to do is to generate a call to a specific function name. In the early days of OOP, this additional overhead was considered a problem, and some languages (most notably C++) provided the procedural-style static binding as the default. However, this also caused problems in practice [117][113][72]. Except in certain domain-specific problems, more importance is presently given to characteristics such as ease of modelling, development and maintenance.

Liskov, in a keynote address given at the OOPSLA'87 conference, presented a desirable property of subtypes relative to supertypes [99]: the ability to use an instance of a subtype in a context where an instance of its supertype is expected. This came to be known of the *principle of substitutability* [80], or the *Liskov substitution principle*. When this principle holds to classes as types, it is possible to create new subclasses and make existing programs to function with them the same way as with the original classes. This follows the open-closed principle and ensures that classes are extensible.

These properties led many software developers in the early days of OO to misguidedly believe that inheritance would enable programs to be extensible as well. Unfortunately, this is far from the truth. The reason is that classes are of limited worth when taken in isolation. A program is not one class, but an arbitrarily complex collaboration of objects.

Inheritance enables a class to extend another class, but not an arbitrary system of classes or a set of methods scattered throughout multiple classes. Dynamic binding enables one method to override a method with an identical signature in the superclass, but not an arbitrary set of fragments scattered throughout multiple methods, in one or several classes. Whenever the functionality we would like to extend is not in the form of a class or a

method, developers must extract the desired unit of modularity to a separate place in the system and restructure the remaining parts so that the new unit is used according to the Liskov substitution principle. For instance, whenever the system does not provide the desired method to be used or extended, developers must extract it and place it in the appropriate class, after which it is available to be called or overridden. In practice, developers identify the useful abstractions to be captured in classes and its respective operations through a discovery process. Johnson and Foote remark in [69] that developers usually discover such abstractions only after acquiring experience with various concrete examples. Frameworks (see section 2.1.5) are developed and matured through this discovery process as well.

Issues of style play an important part in this discovery process. An appropriate style facilitates the development of class systems. For instance, adopting a style favouring small classes and methods makes it more likely that we will find the class or method we would like to (re)use or extend. Johnson and Foote [69] propose a set of style rules, using Smalltalk as the illustrating OO language. These rules are the prototypes for the catalogues of bad smells and refactorings (see 2.2) proposed by Fowler et al. [47], using Java as the illustrating language.

Assessment of OOP

Although software reuse was a much-talked goal in software engineering well before the advent of OOP, the concepts it supported led many to think we finally had a technology that would deliver genuine software reuse. Objects would be "naturally reusable" because of the aforementioned properties. However, experience acquired later showed that this was far from being the case. One manifestation of this disappointment is the statement by Udell that OOP had failed on its promise of software reuse [155], mentioning components developed with a non-OOP programming tool (Visual Basic) as an example of greater success with software reuse. Nierstrasz and Dami (cf. first Chapter of [124]) refer to a debate in the Internet, prompted by Udell's article, which led to the conclusion that reuse depends more on design and methodology than technology. Nierstrasz and Dami expressed the view that developments in both areas are needed for software to become effectively reusable.

Presently, it is generally agreed that OOP provided a better way to organise software, including source code, relative to previous models such as procedural programming. However, two decades of experience with OOP also led to the conclusion that although OOP signalled a significant advancement in software engineering, it still requires an explicit effort to deliver reusable software. Ossher e Tarr [130][131] made a particularly perceptive remark on current state of affairs in software development: "At present, software is like clay: it is soft and malleable early in its lifetime, but eventually it hardens and becomes brittle. At that point, it is possible to add new bumps to it, but its fundamental shape is set, and it can no longer adapt adequately to the constant evolutionary pressures of our ever-changing world."

One of the reasons why this occurs, even with software developed with OO technologies, is that although OOP's support for separation of concerns is still insufficient. Many cases and problems can presently be found that include concerns whose code the object model is not able to keep in separate units of modularity. Moreover, such cases are appearing more often, due to the increasing complexity of software.

These problems occur because OOP shares a limitation with all previous programming paradigms: the support for only one, single criterion for the decomposition of a system into units of modularity. A single criterion becomes the dominant one, and as a result, concerns that need to be decomposed along some different criterion usually become "scattered"

through the design and code. Ossher and Tarr [130][131] dubbed this problem the "tyranny of the dominant decomposition". It is one of the root causes of the scattering and tangling effects.

2.1.2 The Law of Demeter

The *Law of Demeter*, proposed by Lieberherr and Holland [95][93][94], is a style guideline for writing "good" object-oriented code. It basically states that objects in object-oriented programs should deal only with its "closest neighbours", the exact definition of which depends on which of the various forms of the law is being applied. There are "strong" and "weak" forms of the law, and there is also an "object-form" and a "class-form". The weak form of the law states that methods should send only messages to the self-variable, the fields of the self-variable and the method's arguments (these also including objects local to the method and global objects, in case the language's rules allow for its existence). The strong form of the law is similar but excludes from the previous set all accesses to fields inherited from superclasses. These must be mediated through accessor methods.

The aim of the authors of the law was to reduce coupling between objects, so that changes in an object (or, to be more precise, to the code of its class) impacts on the smallest possible set of other objects and classes. If, for instance, the strong form of the law is followed in languages that have the private access mode similar to C++'s and Java's (meaning that only instances of the exact class can directly access those fields), then all fields are declared private and any changes to them bear no impact on the outside of the class, including subclasses. In particular, the law prohibits the deep nesting of accessor methods such as object.getPart1().getPart2().getPart3().getPart4(), because such sequences make the code dependent on the particular structure of the various objects that are included in these sequences (the law allows the nesting of constructor calls, whose use in languages such as C++ and Java is anyway unavoidable).

The class form's versions (strong and weak) are expressed in terms of classes, while the object forms are expressed in terms of objects [93]. The existence of both an "object form" and a "class form" is due to the fact that even if it is the object form that expresses the author's true intentions (the "spirit" of the law), it is not possible for compilers to enforce it at compile-time. However, compilers can enforce the class form, and that is the primary motivation for the existence of the class form.

Despite some benefits that following the law in its various forms may bring, it has been criticised on several fronts (cf. [141] and pages 668-671 of [113]). In particular, Czarnecki and Eisenecker [34] note that appliance of the law leads to object-oriented programs full of small methods that do no or very little computation. Most of these methods simply call other methods, passing information from a set of objects to another set of objects in the same program. Programmers new to the code that try to understand the intents of the original developers are forced to an "endless chase" through such small methods, wondering where the "real" computation gets done. In addition to this understandability problem, Czarnecki and Eisenecker also point out that even when the law is followed, changes in a part of an object structure that is not directly involved in the computation but needs to be traversed by the small information-passing methods require the manual modification or addition of these small methods.

2.1.3 Design Patterns

Design Patterns [48][135][23] are a convention for documenting generic and reusable solutions for problems that recurrently occur in multiple systems and contexts. Design

patterns explore the possibility of using a proven solution to a problem to create similar solutions for other, similar, problems. Each design pattern focuses on a particular design problem, abstracting from context-specific situations and tools, including programming languages, and identifies all the important issues – both positive and negative – of the solution in consideration.

Each design pattern is given a carefully chosen name, which can be used to enrich a vocabulary shared between system developers. This way design patterns contribute to a “bandwidth increase” in their conversations. A collection of design patterns related to common subject is said to constitute a *pattern language*. For instance, [144] presents a pattern language for dealing with concurrency and synchronisation in the development of concurrent OO applications.

Design patterns are not restricted to object technology, like other techniques such as components and frameworks. However, design patterns often are associated with OOP, because they appeared when OOP was starting to gain a significant following, first through OO languages such as Smalltalk, CLOS, C++ and Eiffel, and more recently Java. For instance, all 23 patterns from the most popular book on the subject [48] are based on OO technology.

Design patterns express systems of collaborating classes, their relationships and responsibilities, and thus strongly influence the design of larger structures, such as software architectures and frameworks [17] (see section 2.1.5). Patterns provide the conceptual building blocks of frameworks and that is why there is a close relationship between the two concepts [68]. However, frameworks reside at the source code level, while design patterns are at a higher level of abstraction. Patterns are not meant to be used “as is”, but rather require the developer to reimplement the ideas they embody in every new specific case. Though the pattern documentation often includes fragments of code in specific languages – termed *idioms* – these serve illustrative purposes, and the developer must tailor the solution to her needs. Patterns are a tool for design reuse, but do not deliver reuse at the code level.

Most design patterns, including most of those collected in [48], but also in [23], [143], [144] and other publications, offer solutions for the separation of concerns in situations of various kinds. For each situation, the use of a pattern will prove to be more or less convenient and advantageous depending on the designer’s particular aims. Since many patterns are based on the traditional concepts of object technology, namely inheritance and object composition (association and aggregation), code based on these patterns, including framework code, also suffers from the problems mentioned in section 1.1. In most cases, the use of a pattern involves trade-offs.

In [34], Czarnecki and Eisenecker mention some of the problems associated with the use of design patterns. One of the most serious is the *preplanning problem* [1][34] – architectures that result from applying patterns are usually more flexible for certain kinds of changes, but they also result more difficult to adapt to other kinds (i.e. they become “brittle”). It is the responsibility of the designer to anticipate all changes that might be required of the system during its lifetime, and select the appropriate patterns and respective implementations, weighing the various advantages and disadvantages. Here lies one of the problems with patterns: as we argued in section 1.1, anticipating all future requirements of a system is difficult, and sometimes impossible.

A second serious problem mentioned by Czarnecki and Eisenecker is the fact software compositions in most patterns are based exclusively on the mechanisms of inheritance and aggregation, which introduce more complexity than necessary. This causes *object schizophrenia* [1][34] – the tendency of many patterns to break designs, which would

otherwise preserve their unity and coherence, into several parts (objects and classes) for the sake of a specific separation. This increases the tangling and scattering effects already at the design phase, and leads to a loss of intentionality.

Another problem is the *traceability problem*. Traceability [27][26] measures the ease with which one is able to determine how one of the parts of a system (be it requirement, design or code) affects others. The traceability between design and code in many pattern implementations is poor, due to its indirect representation of design patterns in the code. It is often hard to determine exactly which pattern was used in the design and for which purpose from looking at a given implementation.

The most obvious solution to the traceability problem lies in turning the design patterns into language and library features [24][11], in order to increase intentionality and traceability. This is already happening: for instance, the Iterator pattern ([48], pp. 257-271) is used in the collection application program interface (API) for the Java programming language [51] and the next version of Java (1.5, codenamed "Tiger") [5] subsumes the pattern into the language syntax. Another example is the Observer pattern ([48], p. 293), which is the basis for the Observer and Observable interfaces from Java's standard java.lang API (see also section 7.3). In some cases, the availability of some feature can remove the need for a given pattern altogether. In fact, design patterns could be regarded as a sign of limitations in existing languages. For instance, the decorator pattern ([48], pp. 175-184) is really a technique for emulating mixins [22] in languages that do not have the features necessary to directly support them (such as Java). Likewise, the Visitor pattern ([48], pp. 331-344) is an attempt to deal with two simultaneous criteria for decomposition, which could be dealt with in a more elegant way if a mechanism for multiple dispatch were available.

Czarnecki and Eisenecker view design patterns as an useful technique for documenting limitations in current tools, that can move on to more advanced, possibly higher-level features, as experience is gained and the tools evolve. They also think patterns will retain their usefulness for documenting the applicability of language features for solving specific problems.

Czarnecki and Eisenecker also mention a drawback that design patterns have from the performance point of view. They note (cf. page 61 of [34]) that design patterns typically rely on dynamic binding even in cases where the variation points remain frozen within a single application, in which case static binding and partial evaluation would be more appropriate. For instance, in his Ph.D. thesis (page 159 of [104]), Manolescu states the following when evaluating the OO framework he developed: "The ability to plug into the micro-workflow core components that implement advanced workflow features requires adding hooks to the core's components. Once added, these hooks incur runtime overhead even when the pluggable components they accommodate aren't plugged in. This overhead represents the run time cost of the flexibility achieved through composition." These remarks apply to many complex OO software systems.

2.1.4 Components

Component is a more general concept than that of object. The concept of component [149][161] aims to embody the alluring idea of *software integrated circuits* (software ICs), proposed by Cox [33] in the context of OOP. Nierstrasz and Dami (first Chapter of [124]) proposed a definition of component as a software structure specifically designed to be reused in collaboration with other components. One component taken in isolation does not comprise an application, but a component is independent of any specific application. Components enable their users to abstract from the implementation details, by

encapsulating definitions behind a well defined, independent interface. Concrete applications result from the composition of several components and the instantiation of the parameters passed through their interfaces.

The concept of component can be characterised by stressing the following properties [161]:

- *Components are pre-built.* Components are usually distinct and separate items, such that their identities are not wholly lost in the assembly and, if the assembly is disassembled, they can be recovered relatively intact. Software components retain separate identities within the runtime software of which they form a part because they are pre-built. This reflects the fact that, components are generally made available in binary form, rather than as source code units that might be assembled into a program and then compiled. There are historical as well as pragmatic reasons for this being so. For instance, using pre-built components is cheaper than reusing code earlier in the development process, since it cuts out the earlier parts of the development process. Another pragmatic reason is the fact that organisations generally want applications to be integrated "non-intrusively", or "non-invasively", meaning without code changes. Being pre-built helps to be built with non-intrusive reuse in mind. Maintainability is also a reason: being pre-built places a much stronger boundary between components than there is between classes of an OO application, particularly if the source code is not available. In most cases, this enforces the assumption that a change will impact only the component itself, which in turn reduces the amount of code inspection required and results in lower costs for software development.

- *Components are black box, accessible only via their interfaces.* The ability of components to interoperate with other components, and the way they do this, is also important. Components are "black-box", that their internals are not visible to users, which are interested only in the functions or "services" they offer through interfaces. The design of the component's interface is key to their specification.

- *Components are separable, context-independent.* It must be possible to separate a component from its context and use it in another context.

In [161] a definition is proposed that gathers the above mentioned characteristics of a component: "A software component is a separable piece of executable software that makes sense as a unit, and can interoperate with other components, within some supporting environment. The component is accessible only via its interfaces and is capable of use as-is, after any necessary installation and configuration procedures have been carried out. In order for it to be combined with other components, it must be possible to obtain details of its interfaces."

The above definition recognises that the reusability space of a component is limited to a given environment (i.e. the component built for running on a given environment can be reused only within an instance of that environment). Keeping with Cox's "software IC" analogy, we can say that environments comprise the software IC boards. Examples of such environments can be platforms such as the Java platform or .Net, as well as many frameworks. Each component must make assumptions about its environment. Platform and frameworks serve to narrow the assumptions that a component has to make about its environment to manageable proportions. This does not indicate the order with which components, platforms and frameworks are developed. Sometimes, a component is created during the development of a framework, in which case it is designed to conform to the rules of the framework. In other cases, a pre-existing component is reused in a framework,

in which case the framework must provide the necessary conditions that ensure its correct use.

Unfortunately, experience in software development showed that there are limits to the feasibility of black-box components and software ICs. The IC analogy may even be cause to some misunderstandings, as it may lead some people into thinking that software reuse can be attained by simply mimicking the black-box nature of hardware components. In fact, hardware ICs are comparatively used in much more narrow range of environments than software. The environments of h/w ICs keep most variables fixed for each individual instance, namely voltage, clock speed and number of pins. Therefore, to keep an h/w IC isolated from "changes" in its environment is really a non-issue. Whenever we switch from a board to one with different characteristics, we replace the ICs as well. The new board may contain similar ICs that provide identical functionality and interface, but are prepared to work with the new physical characteristics. The situation is different with software components. An enormous range of elements in the environment keeps changing, including the underlying h/w, the operating system, the libraries, the programs that generate the inputs, the programs that receive their outputs, the programming language used to write other components, etc. Yet, it is expected of software to continuously adapt to all these changes. To require a component to be usable in uni-threaded, multi-threaded and distributed environments is roughly the equivalent of requiring an h/w chip to be adaptable to several different voltages and clock frequencies.

When programmers ponder on the set of elements of the implementation that should be included in the interface to prepare the system for all potential changes in requirements and the environment, they usually find the task self-defeating. As new requirements arise, programmers are inexorably pushed to the conclusion that each new problem demands a fresh set of implementation elements to be exposed in the interface, until there are no implementation elements left to hide. Ultimately, in order to deal with every potential change to requirements and environment, practically everything in the implementation needs to be exposed.

A particularly significant technical hurdle is the fact that interfaces of software components, which in the general case are restricted to the signatures of routines or methods, only cover a narrow subset of the characteristics of the surrounding environments. Programmers do not have the possibility to parameterise characteristics of the environment that often play an overriding importance in the requirements for the software system. This particularly applies to non-functional characteristics (e.g. time constraints or the maximum amount of memory used). In such cases, developers have no other option than to resort to direct intervention in the implementation code.

In [81] Kiczales illustrated the problem with the limits of abstraction and interfaces, as they are currently understood. He presented an example comprising a hypothetical architecture for a spreadsheet application, structured into several layers of abstraction. In his example, each layer would provide the desired functionality while hiding the implementation details from the layer above, and interfaces would comprise the boundaries between two different layers of abstraction. The system would be built on top of window system, which in turn be implemented on top of an operating system, and so on. Each cell of the spreadsheet would be implemented as a window from the window system. The proposed architecture would benefit from the usual advantages of abstraction, including portability and reusability. However, it would be plain to even inexperienced programmers that such a system would either not work, or at the very least its performance would be so bad as to render it worthless for all practical purposes. This would be so because the proposed design was based solely on the principles of abstraction without taking into account others issues that do not show up in interfaces, namely the overall system performance. The fact

that the implementation of the window system would likely fail to give adequate performance would not be deduced from the system's interface, and most likely neither could it be deduced from the system's documentation. In order to address the above problems, Kiczales proposed open implementation, which is discussed in section 2.3.

2.1.5 Frameworks

In order to restrict the dimensions of potential change to a manageable set, a technique that is related to the above – frameworks [42][69][68] – was developed along with components. Frameworks are typically collections of cooperating, and generally abstract, classes that realise a specific architecture and encapsulate all algorithms common to a family of applications. Some of the operations declared by the abstract classes can be left unimplemented because its implementation is specific to concrete cases (cf. the "behaviour classes" or "programs with holes" mentioned by Meyer in pp. 505-6 of [113]).

A framework is a sort of *code template* containing the common code that can be completed by application-specific code, resulting in concrete applications, or *framework instances* [7]. These can be obtained through customisations, usually by integrating in the framework domain-specific classes that inherit from the abstract classes and contain all the missing functionality needed to build a complete application. Frameworks are usually characterised by the *inversion of control* property: abstract classes call the application-specific classes, and not the other way round. This enables the general part of the framework to control and fine-tune the behaviour that domain-specific classes are permitted to override and/or customise.

The above definition of framework emphasises its clear-box nature, in contrast to that of components. Being clear-box, frameworks are more flexible and customisable than components. However, frameworks also force users to learn their architectures and internal structures, which is the cause of one of the problems in using them. Because of its complexity, they are hard to learn and hard to develop. For instance, Johnson points out in [68] that programmers complaining about the complexity of a set of dependent classes are usually a sign that the classes are part of a framework. Frameworks provide the usual trade-off between power and complexity in system development.

As pointed out in section 2.1.2, frameworks share with design patterns the preplanning problem. However, the problem really becomes serious in the kind of large and complex structures that frameworks tend to be (or to grow into being). In addition, the situation often arises in frameworks in which a single class has several superimposed roles from various different patterns [157] (some developers proudly refer to the "pattern density" of their creations [14][157]). This increases the traceability problem we mentioned in the context of design patterns (see section 2.1.2). Lack of traceability makes it harder to foresee the consequences of a given change in the class. This is a consequence of the fact that present OO languages do not provide constructs to express design patterns more declaratively or intentionally.

2.2. Refactoring

Refactoring [184][125][47] is a technique that attempts to deal with the preplanning problem and to cope with the continuous changes in requirements and environments. The purpose of refactoring is to improve the design inherent in the existing code, while maintaining its behaviour. Refactoring is changing a code base in such a way that it corresponds to a better design, a procedure reverses the traditional order of design first, code next. The refactored code is meant to be better organised and easier to maintain,

adapt and extend, while providing the same functionality. A variety of specific code transformations – themselves called *refactorings* – is employed in this transformation process. Each refactoring describes a disciplined way to modify code in order to achieve a specific design change, without introducing compiler errors, bugs, or otherwise affecting the application's "externally observable behaviour"[6] [47]. Chapter 7 describes one refactoring process. The first Chapter of [47] describes another.

Refactorings are typically performed through small steps, often with tests performed in between, to make sure that no errors are introduced. It is considered prudent to perform a given restructuring with a sequence of small refactorings rather than a few large ones, as large restructurings increase the likelihood of introducing errors. Larger refactorings are usually decomposed into several small ones. Sometimes a given refactoring may require several others to be made, before the code is ripe for it to be applied. For instance, when referring to *Extract Method* ([47], p. 110) Fowler et al. mention awkward situations that may require the previous use of two other refactorings before *Extract Method* ([47], p. 110) can be safely applied.

Refactorings can be performed either manually or automatically. Though the earlier research efforts focused on tools that could automatically perform behaviour-preserving transformations on the source code, a more recent approach appeared [12], proposing that such transformations be performed manually [47] in case such tools are absent from the development environment. Nevertheless, the availability of tool support for code transformations is very desirable to provide safety in its use and to increase productivity. In addition, tools generally support the undo operation of the refactorings, something that is important for the situations when the programmer finds out that she took a wrong turn.

Refactorings are not meant to introduce new functionality. Code transformations that add or change existing functionality are not considered refactorings. This is sometimes represented by Beck's "two hats" metaphor ([47], p.54). The programmer is always performing one of two distinct activities: adding function or refactoring. She is either wearing the developer hat or the refactoring hat, but not both. She may be adding a piece of functionality, in which case she is wearing the developer hat. She may realise that her task would be much easier if the code were structured differently, so she swaps hats and refactors for a while. Then she swaps hats again, and adds the functionality. She may swap hats frequently during the software development process, but at every moment she should be aware of which hat she is wearing.

Refactoring is different from optimisation. Optimisation is usually a behaviour-preserving change to a program with the goal to make it more efficient. Such changes often make the source code less intentional, which is the opposite of what is aimed at with refactoring. Fowler concedes that changing a program to make it more intentional does yield slower programs, but also tends to make programs more amenable to performance tuning. Fowler also notes that the majority of programs spend most of its time in a small fraction of the code. He warns that to blindly strive for good performance in all parts of the system tends to result in programs that are harder to work with, which brings its own set of costly consequences. Fowler states that, excluding exceptional cases where this fact does not apply, developers should write tuneable software first, and then tune it for sufficient speed.

[6] Since one of the key requirements of refactoring is to preserve behaviour, it does not make sense to apply refactorings to a code base that gives rise to compiler errors, for such code bases do not have a concrete and testable behaviour one can think of preserving.

2.2.1 Role of Units Tests in Refactoring

Refactorings are often described as "behaviour-preserving transformations of source code". The most often used mechanism to express the desirable behaviour is tests. As Fowler states in [47] (p.89), "if you want to refactor, the essential precondition is having solid tests". Without tests, there is no way to know if the desirable behaviour of an application changed or not.

The tests normally associated with refactoring are *unit tests* – highly localised tests dedicated to testing a single unit of modularity, usually a class or method. There are other kinds of tests. For instance, *integration tests* are written to test a collaboration of objects works as it supposed to. *Functional tests* are written to ensure that a whole subsystem or the system as a whole works as planned. *Stress* or *load tests* serve to assess whether an application performs well even during particularly demanding conditions (e.g. when there are many people using it, or when it must process a large number of requests within a short period of time). Finally, there are *acceptance tests* to assess whether the application actually meets the customer's needs. Acceptance tests are usually conducted directly by the costumer or someone acting as the costumer's proxy.

Currently, a large and growing community of software developers manage suites of unit tests using tools such as the JUnit framework [183]. Such tools are becoming so widespread that are being integrated with some of the existing IDEs. For instance, eclipse [179] includes JUnit in its standard installation package.

One important feature of the tests developed according to frameworks such as JUnit is the fact that the programmer is not required to look at console or printed outputs to check whether results are as they were supposed to be. Instead, the tests themselves carry out the relevant comparisons – they are *automatic*, meaning that they notify the programmer of results only if problems were detected. Otherwise, the tests terminate silently. In addition, in tools such as JUnit it is possible to execute an entire collection of tests with a single instruction. JUnit is capable of automatically identifying all the tests within a class or package and executing them without requiring further intervention on the part of the programmer.

It is important that tests are automatic, because otherwise the number of tests that can be performed in sequence would be severely limited, and using them would be an exhausting task that people would be reluctant to perform frequently. On the other hand, the number of automatic tests in a system can be arbitrarily large, and that is why it is essential that tools such as JUnit be able to automatically run all tests placed within the specific class, package or subsystem. Programmers need to worry with the results only when one or more of the tests fail, giving rise to JUnit's famous "red bar". Figure 3 shows an example, taken from [15]. Use of automatic tests also seems to lead to a more psychologically satisfying approach to testing [16] than with previous, conventional, ways, opening the way for the *test-driven development* approach to software development [4][13].

Unit tests also serve as r*egression tests*, i.e. tests that ensure that new modifications do not introduce fresh bugs into code that was previously passing all the tests. Thus, unit tests blur traditional boundaries between the phases of coding, testing and debugging.

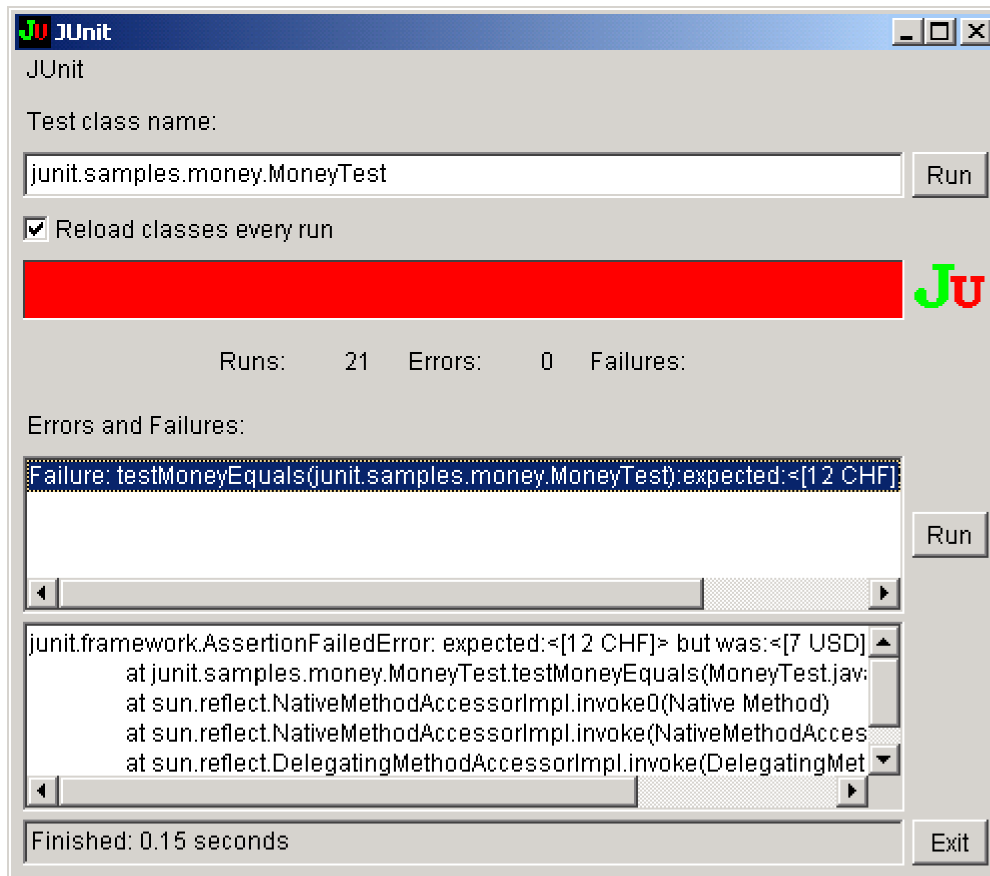

Figure 3: JUnit's red bar signalling a failed test.

2.2.2 Brief History of Refactoring

Though programmers have been applying source code transformations for decades, only more recently has refactoring been subject to formal research. One of the earliest works is Griswold's doctoral work. In his PhD thesis [53] and again in a paper co-authored with his thesis supervisor, Notkin [54], Griswold remarks that unanticipated changes are likely to cut across the module structure of the system, and that the resulting needs for non-local changes are the key property of structural degradation and one primary cost factor in software maintenance. The authors claimed that the global restructuring of software systems could be cost-effective, if it is automated and separated from other qualitatively different maintenance activities. Griswold focused on automating a set of program transformations that are difficult for the human programmer to perform correctly or quickly, but are easy for the computer to perform. The automation of such transformations was meant to free the software engineer from those tasks, enabling her to concentrate on the subjective activities of choosing the appropriate structure.

Griswold focused his work on procedural and functional programs, and included a prototype tool for restructuring Scheme programs. Opdyke's Ph.D. thesis [125], under the supervision of Ralph Johnson, is generally considered the first major written work on the subject of automatic refactoring of object-oriented programs. Opdyke and Johnson were also the first to use the term "refactoring" in the context of transforming OO programs [127].

Opdyke identified a series of refactorings for object-oriented source code, using C++ as the subject language. Opdyke described the structural prerequisites and automatic program transformations necessary to guarantee preservation of behaviour [125]. Opdyke and

Johnson later defined various refactorings for C++ [126][70]. Similarly, Tokuda and Batory evaluated the impact of a refactoring tool for C++ [151][152]. Though Opdyke's work focused primarily on C++, it opened the way for the first widespread refactoring tool – the Refactoring Browser [137]. This tool was developed for Smalltalk by John Brant and Don Roberts, also under the supervision of Johnson. In his Ph.D. thesis [136], Roberts specifies the refactorings provided by the Smalltalk Browser and focuses on the possibility to combine refactorings by analysing post conditions and preconditions of the combined refactorings.

Since the end of the 90's, refactoring became the subject of growing interest due to the advent of agile methodologies, most notably *Extreme Programming* (XP) [12]. In this context, refactoring is regarded in a subtly different light. Earlier research was based on the assumption that automatic tool support is a fundamental prerequisite for refactoring to be feasible. Therefore, the advent of refactoring tools would open an entire new range of possibilities. From the point of view of advocates of XP, however, those possibilities can be realised even without the support of refactoring tools. The availability of testing frameworks such as JUnit [183] made the difference. Naturally, advocates of XP welcome tool support for refactoring as a very important contribution to enhancing their productivity. These views are clearly expressed in the book by Fowler [47], which also acts as a manifesto for the use of refactoring in software development and maintenance.

Fowler et al. [47] present a manual but disciplined approach to the use of refactoring. The primary component in this book is a collection of 72 manual refactorings for object-oriented programs (see Appendix A). The subject language used in Fowler's book is Java, and the book was published at a time when tool support for refactoring Java programs was not available. The refactorings were meant to be performed in a manual but disciplined way. After publishing his book, Fowler challenged tool vendors to produce automated refactoring tools for mainstream languages such as Java. Fortunately, the tool vendors responded, and few years later several IDEs provided automated support for many of the refactorings described in Fowler's book.

Presently, Fowler maintains a web site with material on the subject of refactoring [184]. The site does not provide many of the contents of his book, but includes a list of refactorings, including some that were described after the book's publication (see Appendix C). There is also a very active mailing list to discuss issues related to refactoring and style [185].

2.2.3 Bad Smells in the Code

Having tool support for refactoring is an important though not sufficient factor. Presently, the available tools can already automate many of the most widely known refactorings, but cannot automate the thought process that leads to the decision to apply a given refactoring. Though most of the desirable code transformations may one day be completely automated, the process that leads to decisions of when to refactor, what to refactor, and how to refactor is expected to remain a subjective one and therefore the preserve of human programmers. Humans must make the decisions, and for that they need rules of style. A clear notion of style enables programmers to recognise the situations in the code that require intervention.

Notions of style in a programming language are meant to express the coding practices that result in code easier to understand and to evolve. This entails making the code communicate the intentions of the programmer as clearly as possible (i.e. favouring the code's intentionality). Refactoring, like XP, regards a system's source code as primarily a communication mechanism between people, not computers. Whenever a programmer has

difficulties in understanding what a snippet of code is supposed to do, this is considered a sign that the code should be refactored. This is illustrated in Figure 4, where the motivating example for the *Compose Method* ([76], p.123) refactoring is shown.

```
public void add(Object element) {
   if(!readOnly) {
      int newSize = size + 1;
      if(newSize > elements.length) {
         Object[] newElements = new Object[elemnts.length + 10];
         for(int i=0; i<size; i++)
            newElements[i] = elements[i];
         elements = newElements;
      }
      elements[size++] = element;
   }
}
```

```
public void add(Object element) {
   if(readOnly)
      return;
   if(atCapacity())
      grow();
   addElement(element);
}
```

Figure 4: Introdutory Example for the *Compose Method* refactoring.

In the days of procedural programming, the original form of the example shown in Figure 4, which includes three nested control structures within the same method, would be considered standard practice. In the light of the style proposed in [47][76], it smells of the *Long Method* smell ([47], p.76). Figure 4 also illustrates the tendency of refactorings to enrich the interface of the refactored objects. Though Fowler's catalogue also includes refactorings to inline methods and classes, more often than not the refactoring process yields a system with a larger number of finer-grained units of modularity. This is in keeping with the interface-oriented and interface-driven nature of OO.

Fowler's book contributes with a more developed mechanism to express notions of style. Fowler explains in Chapter 3 of his book [47] that describing the situations that lead to the use of given refactoring is trickier than describing the mechanics of the refactoring itself. The strategy he adopted, on a suggestion by Beck, was to describe what the code *should not* be, instead of describing what it *should* be, through the *bad smells in code* metaphor. Code smells are a way to describe warning signs about *potential* problems in code. Not all smells indicate a problem, but most are worthy of a look and a decision of whether to refactor or not to refactor. Fowler's book includes the description of 22 different smells, and indicates the refactorings that are likely to remove them from the code (see Appendix B). The collection of smells serves as a refactoring guide for programmers.

A decade earlier, Johnson and Foote provided "rules of thumb" [69], for obtaining well-structured OO classes that are amenable to reuse. These rules can be considered prototypes of the rules proposed by Beck and Fowler, touching issues of both design and style. The differences between the rules by Johnson and Foote and those proposed by Fowler et al. are in the subject language (Johnson and Foote use Smalltalk) and in the greater maturity and level of detail provided by Fowler and Beck. For instance, Foote and Johnson propose a *Reduce the number of arguments* rule that corresponds to the *Long Parameter List* smell ([47], p.78), but also to the refactorings *Extract Method* ([47], p.110), *Extract Class* ([47], p.149), *Preserve Whole Object* ([47], p. 288) and *Introduce Parameter Object* ([47], p. 295).

In his workbook on refactoring [158], Wake elaborated on the smell metaphor by evoking the image of someone opening a fridge that has a few things going bad inside: “some smells will be strong, and it will be obvious what to do about them. Other smells will be subtler; one won’t be sure if the problem is caused by this or that dish from yesterday. Some food in the fridge may be bad without having a particularly bad smell.” Wake states that “code smells seem to be like that: some are obvious, some aren’t. Some mask other problems. Some go away unexpectedly when one fixes something else”.

Wake contributes with more smells (see Appendix F). In addition, Wake introduces the concept of *symptom*, which is closely related to that of smell, but mentioning a concrete situation instead of providing a general description. In addition, Kerievsky published a book documenting 27 refactorings to introduce (and sometimes to remove) design patterns in existing code [76] (see Appendix D), and contributes with several additional smells (see Appendix E). The refactorings address both the new smells he proposes as well as some of the smells described by Fowler et al. Kerievsky’s refactorings are presented in a format similar to that used by Fowler, and we also use in Chapter 6.

2.2.4 Why do we need Refactoring?

Ideally, programmers would be able to place all concerns in a software system in their own units of modularity, and would be capable to perform any compositions of those units according to the conveniences of each specific case of use. In practice, programmers are not able to do that, because traditional programming models and tools are limited in terms of the composition mechanisms provided and units of modularity supported. Consequently, whenever there is a need for an unsupported composition, the system’s source code must be manually changed to derive something equivalent to the desired composition.

Existing composition mechanisms support only a limited set of units of modularity, and changes in requirements and environments entail changes in the existing units. Whenever programmers need to perform compositions over code not confined within the available units, programmers are forced to first extract new units from the code base. In systems subject to evolving requirements and environments [133][56], this process may need to be continuous and permanent. Limitations on the compositional capabilities also have the consequence that existing units tend to contain code related to more than one concern (giving rise to the code tangling problem). Each time that changes must be made to one of the concerns, it is necessary to change the unit anew. This consequently increases the rate at which a given unit must be subject to changes.

Another cause for the need to refactor is the fact that many programmers write OO code in poor style [113][47]. One of the likely causes for this is the *paradigm shift* problem [88], i.e. the fact that absorbing the concepts and way of thinking of OOP seems to be much harder to people previously used to the procedural style. Anecdotal evidence (e.g. page 172 of [148]) suggests that it takes programmers from a half year to one and a half years to switch the mindset from a procedural to an OO view of the world. On the other hand, experience also suggests that OOP is more easily absorbed by people that did not have a previous contact with procedural programming. It is the switch that is difficult, not OOP itself. Judging from the examples of bad style presented in [69][47][76][158] and elsewhere, the mental misadjustment caused by a procedural background seems to be a major cause of badly written OO code.

2.3. Metaprogramming and Reflective Techniques

There are many tools that automate a wide variety of tasks in software engineering, including compilers, type-checkers, interpreters, debuggers, translators, code generators, optimisers, and profilers. Recently there have been new additions, namely aspect weavers (see section 2.4) and refactoring tools (see section 2.2). There is a common element in all these tools: they use representations of programs that are processed in some way, and that can take various roles in that processing (as input, as output, or both). All these tasks fall under the umbrella term of *metaprogramming*.

Briefly, metaprogramming [34] is the automation of programming, the activity of developing programs that read, manipulate, and/or write other programs. It is characterised by the fact that the representations being processed do not relate to a particular problem domain, but to the programs themselves. In the special case in which the representation the metaprogram is processing relates to *itself*, the term computational reflection [103] or simply reflection [87] is used.

In earlier days, the primary mechanism to deal with crosscutting concerns was through language features, i.e. the intervention of the compiler. Crosscutting concerns span an entire system and therefore require a global view of the application, available in an appropriate representation. Decades ago, the compiler was one of the few entities to possess this global representation[7]. Type checking is an example of a crosscutting concern that compilers have been addressing effectively since the days of procedural languages. However, at the time compilers could address only a limited range of crosscutting concerns, for a number of reasons. Compiler technology was less developed, computational resources were more expensive and limited, and anyway it is not desirable that programming languages be changed frequently. In addition, language features can only address general issues, as using language features to deal with case-specific issues is often not practical.

As related technologies developed and matured, new possibilities to address crosscutting concerns appeared, and reflection was a particularly promising trend. In [103], Maes defines reflection as the ability of a software system to query, inspect and manipulate representations of itself. The most straightforward way to implement reflection is through interpreted languages. To support reflection, the interpreter must maintain a representation of the interpreted program, and expose all or part of it to the program itself, through some suitable protocol [87]. This protocol enables the program to perform computations on itself and therefore change its own structure. Through reflection, users of the reflective language would be able to tailor it to the extent that the protocol would allow. Variants of the default behaviour provided by mechanisms such as inheritance, delegation and protection could be specified, and novel kinds of compositions could be obtained [46].

In section 2.1.4, we mention the limitation of traditional approaches to interfaces, which do not take into account important issues such as memory consumption and efficiency. These limitations were pointed out by Kiczales in [81], which proposed a new approach to address these problems that Kiczales called *open implementation*. Open implementation was based on the work by Kiczales and others on meta-object protocols [20][87]. Open implementation proposes using reflective architectures to provide developers with a second interface in addition to the traditional interface [85]. The primary interface would be similar

---

[7] Tools such as preprocessors were another, but they did not prove to be a satisfactory solution. One of the reasons is precisely because the actions of a preprocessor bypassed the compiler and therefore its global representation of the program. For instance, the symbols defined by the preprocessor cannot be type-checked by the compiler.

to that of traditional systems and would relate to the desired behaviour of the system. The secondary interface would be optional and would enable the customisation of particular aspects of the underlying system's implementation to make it better meet the system's needs. Therefore, the secondary interface would deal with meta-level issues.

Despite the early promise, the reflective approach also proved to have drawbacks. Some of them were known from the start, while others surfaced as experience was gained. The first drawback is that reflection requires a reflective architecture, which is expensive. Some languages are often selected for their efficiency and therefore it does not make much sense to develop an expensive reflective infrastructure for them. This particularly applies to languages such as C and C++, which possess characteristics, such as direct access to arbitrary memory locations that are either hard or expensive to reconcile with a reflective architecture. Reflection can also give rise to security issues in some languages. For instance, Java has reflective facilities, but because Java was designed for communication through the Internet, its reflective interface was designed to be "read-only" and to not allow programs to modify their structure and behaviour [25].

In addition, the parts of the source code dealing with the meta-level lack intentionality. The statements comprising the meta level look like those from the base level (unless a different sublanguage is used for the meta-level) and therefore it is hard to tell them apart. Listings 1-3 present a simple example. Listing 1 shows a simple class Capsule defining a simple doSomething method. Listings 2-3 show two client classes that have the same externally observable behaviour. Listing 2 shows a client SimpleCaller calling that method through the standard, straightforward way. Listing 3 shows a ReflectiveCaller client calling the same method through reflection. Listing 3 makes it harder to know at a glance what the intentions are than Listing 2.

On the other hand, some meta-level sections of a program relate to modifying and customising language mechanisms whose default behaviours are often expressed declaratively. These meta-level sections nevertheless have a programmatic form. To have parts relating to the same concern expressed in two different ways is confusing and undesirable. The programmatic nature of these meta-level sections also makes them harder to reason with, and harder to control.

The use of reflective techniques is nowadays regarded as bad-style, i.e. something to be avoided whenever reasonable alternatives are available. One reason for this is the aforementioned lack of intentionality: it often comprises an alternative means to achieve an effect that is already supported through more direct (i.e. intentional) means. For instance, the reflective facilities of Java enable the call of a method through a string containing the method's name. However, this comprises a much more complicated way to call a method than the standard one. In addition, such code is likely to escape the notice of refactoring tools. If, for instance, *Rename Method* ([47], p.) is applied to method doSomething shown in Listing 1, the tool will likely fail to update the reflective call made in ReflectiveCaller shown in Listing 3. This comprises another reason for considering the use of reflection to be bad style.

```
public class Capsule {
   public void doSomething() {
      System.out.println("doing something...");
   }
}
```

Listing 1: Simple class defining a simple method

```
public class SimpleCaller {
   public static void main(String[] args) {
      Capsule capsule = new Capsule();
      capsule.doSomething();
   }
}
```

Listing 2: straightforward way to call a method

```
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;

public class ReflectiveCaller {
   public static void main(String[] args) {
      Capsule capsule = new Capsule();
      Class classObj = capsule.getClass();
      Class[] classes = new Class[0];
      try {
         Method method = classObj.getDeclaredMethod("doSomething", classes);
         method.invoke(capsule, null);
      } catch(NoSuchMethodException e) {
         System.err.println("Class object failed to find the method.");
      } catch(IllegalAccessException e) {
         System.err.println("Acces to method considered ilegal.");
      } catch(InvocationTargetException e) {
         System.err.println("Method invokation failed.");
      }
   }
}
```

Listing 3: Calling a method through reflection

2.4. Aspect-Oriented Programming

AOP is a project whose development started at the Palo Alto research Centre at Xerox, under the direction of Kiczales. At the end of 2002 that project was transformed into an open-source project hosted by IBM's eclipse IDE project [179]. That project (along with other projects such as AJDT [169]) aim to develop a mature software development for the AspectJ language, the primary representative of AOP technology.

AOP is one of several lines of research included in the umbrella term Advanced Separation of Concerns (ASoC) [40][44][170]. The common aim of ASoC is to develop models and tools that enable a greater level of separation of concerns than those attained by traditional technologies such as those surveyed in section 2.1. Unfortunately, ASoC is also referred as Aspect-Oriented Software Development (AOSD), therefore risking being confused with AOP. For this reason, this thesis always uses AOP to refer to the technique described in section 2.4.1 and uses the term ASoC in preference to AOSD.

ASoC also includes a growing number of research lines and projects, namely Composition Filters [176][19][2], Adaptive Programming [96][92], Subject-Oriented Programming [63], Multi-Dimensional Separation of Concerns [150][131][129] and the research conducted by Don Batory [10][8][9]. Some of the aforementioned projects are of limited interest from the practical point of view. For instance, Subject-Oriented Programming is no longer under active development, having being superseded by Multi-Dimensional Separation of Concerns. Composition Filters lacks a modern tool providing a full support of its model. Besides AOP, none of the aforementioned ASoC research projects yielded until now a mature enough a language or tool to be subject to the kind of research focusing on issues of style that was undertaken for this thesis.

2.4.1 What is AOP?

AOP is above all a modularity technology, whose focus is to enable the clean modularisation of crosscutting concerns. AOP was created under the assumption that no programming model or language based on a single criterion of decomposition – be it function, assertion, class or any other – is able to separate and encapsulate all concerns found in the more complex examples of modern software [86]. When a system has two or more decompositions such that neither can fit neatly into the other, one of the decompositions tends to be dominant [150]. Only the elements relating to the dominant decomposition are cleanly modularised: the code relative to others concerns tends to be crosscutting, i.e. they are scattered throughout the modules of the existing structure. Those elements usually appear as code fragments enclosed within the existing modules, i.e. they are tangled with code related to other concerns. In the light of AOP, it is very desirable that models and tools for software development be able to present each concern in the form that most facilitate its understanding, reasoning and maintenance, thus providing greater intentionality. Code scattering and code tangling are an impediment to achieve it.

Kiczales et al. propose a new programming technique they dubbed aspect-oriented [86] as a solution to the crosscutting effect and associated problems. Kiczales et al. use the term *aspect* to refer to concerns or design decisions difficult or impossible to capture cleanly, no matter which programming model is used. AOP enables the representation in its own unit of modularity of code relative to concerns that would otherwise cut cross several units. This new type of module – the aspect – is a representation of a (otherwise crosscutting) concern cleanly captured in its own unit of modularity. The various aspects are then composed through a process dubbed *weaving* [86][64], which produces the application through composition of all the intended aspects.

Aspects are composed by weaving aspect code with a base code. This process works by inlining the instructions from the aspect code into the base code, producing a tangled version. The fact that it is tangled does not impact negatively the development process, since it can be regenerated every time a new version is required. The inlining analogy used to describe weaving may suggest the processing and generating of *source* code, but the earliest aspect-oriented compilers started to perform this inlining at the binary level, very soon in the technology's developing process. This made it clear that weaving could be regarded as one more phase in the execution of a compiler (see Figure 5), and that is how it is presently regarded [6]. There is nothing in the aspect-oriented model stating that the target of the weaving process must be source or binary code, or that the weaving process must be static (compile-time), dynamic (runtime), something in between (e.g. load-time), or supported at the virtual machine level [134].

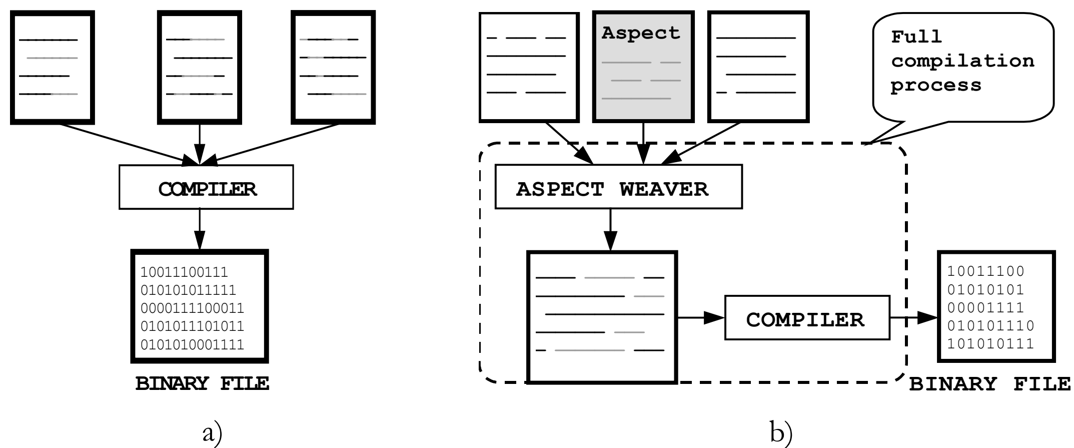

Figure 5: (a) Traditional Compilation (b) Compilation with Weaving

AOP is meant to complement other models, not to replace them. What distinguishes aspects from traditional modules is the crosscutting nature of the concerns they modularise. It is important to realise the relative nature of this concept: a given concern may be crosscutting using one model but may be a cleanly encapsulated one using another. For instance, the procedural model decomposes systems according to algorithms, and therefore data is scattered across multiple units of modularity (procedures). In OOP, data and functionalities comprise the primary decomposition of the system and therefore data is cleanly modularised. Other, non-functional, concerns stand out as crosscutting. That is why the aspects that complement OO applications tend to be non-functional.

Often, the various aspects relate to different abstraction domains. Recognising this fact, the earliest proposals presented by Kiczales et al. include aspects coded in different, sometimes concern-specific, languages. This is consistent with the idea of maximising intentionality. It is the task of the aspect weaver to compose the various aspects into a single application. For instance, Lopes developed two concern-specific languages – COOL and RIDL [100][102] – designed to express policies of synchronisation in multithreaded environments and the management of access to remote objects throughout networks. Although DSLs alone yields maximum intentionality to model specific domains, use of DSLs is also very limiting, since it would require a new DSL as well as its own specific aspect weaver for each new domain, and possibly for each target application language or platform. These problems are not present in the AspectJ language, which is general-purpose.

In its earliest years, there were debates on the nature of AOP and what differentiated it from other models. Filman and Friedman [45] proposed *quantification* and *obliviousness* as the defining properties of aspect-orientation. Quantification of a piece of code over a program is the ability to specify a set of separate points in the execution of the program and then execute the piece of code in every point of the set. In other words, quantification is to say, "whenever condition C arises, perform action A". The kinds of quantifications that an aspect-oriented language supports determine the language's expressive power. A usual quantification can be "whenever method M is called", but can be something more fine-grained like "whenever variable v is accessed for reading".

Obliviousness is the ability to quantify a piece of code over programs that were written oblivious to this code. The programs are said to be oblivious because they were not specifically prepared to receive those quantifications (the programmers who wrote the program were oblivious to them). Thus, it becomes possible to add, or change, or even

delete behaviour from a program without changing its code. Therefore, AOP opens the possibility to reuse code not originally intended to be reused, or in ways other than those initially intended.

2.4.2 Concepts and Mechanisms of an Aspect-Oriented Language

The main aspect-oriented tool is AspectJ [83][82][90], a backwards-compatible extension of Java[8]. Various aspect-oriented tools have later appeared [77], but the majority is strongly influenced by the design of AspectJ. Currently, AspectJ is a de facto standard for AOP in terms of language design. For this reason, and for the sake of simplicity, we use the terms AOP and AspectJ interchangeably throughout this thesis.

AspectJ was designed to be independent of any specific implementation, so that widely different approaches could be taken to support its mechanisms. For instance, an AspectJ weaver can take source code as input, or binary code, and its output can be source code (in which case will have to be compiled again by another compiler), or will directly output binary code. Depending the specific implementations, the weaving process can take place at compile-time, run-time or class load-time, or be performed by the virtual machine itself. The first implementation developed at Xerox (AspectJ 1.0.x) included a weaver that accepted the source code of the target classes as input and directly generated bytecode-compatible binaries (which can be interpreted by any Java runtime). Later, a second implementation (AspectJ 1.1.x) was developed that supports bytecode-weaving, so both input and output come in binary form.

Each different implementation of AspectJ carries its own specific strengths and limitations. For instance, the fact that AspectJ 1.0.x was based on source code meant that weaving could only be performed on the classes whose source code was available to the compiler, a limitation that does not apply to AspectJ 1.1.x[9]. On the other hand, the bytecode-based implementation of AspectJ 1.1.x is facing its own technical hurdles: for instance, the end of exception handlers is undefined in bytecode, which means that AspectJ 1.1.x currently does not support around and after advice on exception handlers. The AspectJ language specification (accessible from [172]) was designed to abstract from these implementation issues.

It falls out of the scope of this thesis to provide all the details of a language as rich as AspectJ. We next describe the primary concepts and present a few illustrating examples (see also appendix G for a quick reference).

Joinpoints

The kinds of quantifications that an aspect-oriented language supports are determined by its *joinpoint model.* Joinpoint is the novel concept that makes oblivious quantification possible. Joinpoints are those elements of the component language semantics with which the aspect programs coordinate. A joinpoint is any identifiable execution point in software system. The following execution events are all examples of joinpoints:

[8] At the time of writing this thesis, the new version of Java, "Tiger" [5], came out. The latest version of AspectJ to be used for the work for this thesis is AspectJ 1.2, which is not compatible with Tiger. AspectJ 5 [173], a new Tiger-compatible version of AspectJ, is under development.

[9] This in turn opened a legal issue, the possibility of weaving third-party code in violation of the terms under which that code is made available. It is usual for software developers to use licenses requiring one not modify the code of the libraries they provide, in part to ensure that they run as expected and tested, and weaving aspects into their code is often a violation of those restrictions. The AspectJ community hopes that in future library licenses start to include permissions to weave aspects, at least in such cases when the aspects do preserve the behaviour of the original code.

- The call to a method;
- The execution of a method;
- The assignment to a variable;
- The access to a field for reading;
- The access to a field to modify its value;
- The execution of the return statement;
- The construction of an object;
- The execution of a control structure (conditional or iterative);
- The condition performed in the context of a control structure;
- The execution of an exception handler;
- The loading of a class by the virtual machine;

Not every joinpoint is supported by AspectJ: only those points that were judged usable in a disciplined and principled manner are. For instance, line numbers could comprise a joinpoint. However, they would be extremely fragile, i.e. they would be very vulnerable to small, otherwise trivial changes to the source code. Another example, presented by Kiczales in a posting[10] to the aspectj-users mailing list [174], concerns a joinpoint on the evaluation of arguments in method calls, which AspectJ does not support. Kiczales notes that most programmers feel that if they the take code like the following:

```
foo(a + b + c());
```

and change it to

```
String temp = a + b + c();
foo(temp);
```

the code would not change its behaviour. However, if the system included an aspect dependent on the evaluation of parameters, the behaviour could change in perhaps surprising ways. This kind of fragility was avoided during the design of AspectJ's joinpoint model. For similar reasons, many other joinpoints are not supported by AspectJ, including several of those above mentioned as examples (appendix G presents those that are). These include the execution of return and those that take place in the internal implementation of methods, such as control structures, comparisons and the use of local variables (access to fields, however, are supported).

Pointcuts

AspectJ includes a novel construct, the *pointcut designator*, which identifies particular joinpoints by filtering out a subset of all joinpoints of a program flow. The joinpoints captured by a pointcut designator correspond to non-contiguous places in the program's source code, and that is why the joinpoint set is crosscutting. Pointcut expressions can include wildcards and thus specify an open-ended set of joinpoints. Table 1 shows a few legal examples of AspectJ's pointcut protocol. Appendix G presents a condensed list of the aspect-specific constructs of AspectJ, including the supported pointcut designators.

One important characteristic of AOP is that joinpoint models are not restricted to elements exposed by the interface as declared by a class. For instance, the joinpoint model of

[10] http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg02826.html

AspectJ enables the quantification of the access of fields (both for reading and for writing), which in most object-oriented systems are usually kept hidden from the exposed interface. For instance, if an AspectJ pointcut is defined to capture all the calls to methods whose name start with "set", without specifying the visibility of the method (i.e. whether the method is public or private), it will capture all calls, regardless of visibility of the methods concerned. This is yet another way in which the crosscutting nature of aspects is apparent, and it somehow suggests that aspects conceptually reside at a meta-level, relative to the remaining "code base".

AspectJ provides 17 primitive pointcuts (these are shown in appendix G) that collect joinpoints based on names and properties. Pointcuts may be combined using set operators such as

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

From all this, it is apparent that AspectJ's pointcut protocol effectively comprises a DSL for one kind of metaprogramming – quantification.

`call(void Foo.m(int))`	A call to the method with signature void m(int) in class Foo
`call(void m*(int))`	Calls to void methods whose name starts with an 'm' and which have a single parameter of type int
`call(* m*(..))`	Calls to methods of any type whose name starts with 'm' and with any nº of parameters, including zero
`execution(!public Foo.new(..))`	The execution of any non-public constructor of Foo
`initialization(Foo.new(int))`	The initialisation of any Foo object that is constructed with constructor Foo(int)
`staticinitialization(Foo)`	When the type Foo is initialised, after its class being loaded by the virtual machine
`get(int Point.x)`	When an access for reading the integer field x in the Point class takes place
`set(!private * Point.*)`	Access for writing the value of any non-private field in the Point class
`handler(IOException+)`	When an IOException or one of its subtypes is handled with a catch block (without the +, type patterns match only the exact type)

Table 1: Examples of pointcuts in AspectJ.

Advice

The constructs that specify how the behaviour of the base program is to be affected are the *advice*. Advices are nameless method-like blocks associated to a given pointcut that execute *implicitly*, whenever one of the joinpoints collected by the pointcut is reached. Note that although advices resemble methods (or procedures) in some ways, they do not have call semantics: advices are never called explicitly, and therefore do not need a name. For the same reason, they are not polymorphic.

Advice can define and use their own temporary variables the same way as methods, and can use and modify whatever values are exposed by the advice's signature, the same way methods can use and modify values received as parameters. Relative to the execution of a joinpoint, advice can execute:

- *Before a joinpoint.* The advice executes immediately before the joinpoint is "entered" and it cannot prevent the joinpoint from executing (unless in abnormal situations, such as throwing an exception).
- *After a joinpoint.* The advice executes immediately after a joinpoint has finished, and therefore does not prevent it from executing. There are variants for joinpoints terminating normally and exiting by throwing an exception.
- *Instead of a joinpoint.* The original joinpoint is executed only if the advice explicitly calls it, using the *proceed* keyword. This mechanism is somewhat similar to calling an overridden method using the super pseudo-variable. This advice can emulate the other two kinds of advice, but it is also more complex to use. Unlike the other kinds of advice, the around advice must declare a return type, which must conform to the type of the joinpoint which triggered it (and which can be void).

Pointcuts can capture context from the joinpoints they collect and pass it to advices, using some kinds of pointcut designator specifically designed for that purpose. AspectJ provides several such pointcut designators, as well as pointcuts to restrict the scope of applicability of the pointcut (see Appendix G). The context passed to the advice must conform to the advice's signature. Table 2 shows a few examples context-capturing pointcuts and Table 3 shows examples of advice acting on those pointcuts.

Pointcut	Description
`pointcut increments(int a):` `call(* incr(int) && args(a);`	Pointcut increments collects all calls to methods incr receiving an int as argument. Parameter 'a' is bound to the value of the argument

`pointcut call2server(Server callee):`
`call(* service(..))`
`&& target(callee);`
	Pointcut call2Server collects all calls to method service of class Server. The instance of Server is bound to the parameter callee

`pointcut fooExecution(Bar self):`
`execution(* foo(..))`
`&& this(self)`
	Pointcut fooExecution collects all executions of methods named foo of any return type and any number of parameters. The currently executing object (i.e. the one referenced by the self variable 'this') is bound to the parameter self.

Table 2: Examples of context capture by pointcuts.

```
before(int v): increments(v) {
   //...
}
```

```
after(Server server): call2server(server) {
   //...
}
```

```
Bar around(Bar current): fooExecution(current) {
   //...
   return //some value of type Bar
}
```

Table 3: Examples of context capture by pointcuts.

AspectJ was designed in such a way as to ensure that the concepts of joinpoint, pointcut and advice are orthogonal. Thus, the definition of a pointcut does not depend on the kind of advices that act on their joinpoints, and a given advice acts similarly on all kinds of joinpoint. Likewise, joinpoints do not depend on the pointcuts that match them or on what kind of advice act on them. If this orthogonality did not exist, programmers would need to remember that a particular kind of advice only works on a particular subset of pointcuts or on a particular range of joinpoints. Such a language would be more complicated to use.

To illustrate the use of pointcuts and advice, consider the simple example shown in Listing 4. When the class HelloWorld is compiled without the Demo aspect, the traditional behaviour occurs and expected and the message “Hello, World!” is sent to the console. When the Demo aspect is included in the build, running the main method of HelloWorld yields the following output:

Before the greeting.
Hello, world!

We usually want to restrict the scope of the pointcuts so that the associated advices do not apply to more joinpoints than initially intended (aspects with runaway pointcuts of this kind are sometimes called *leaking aspects*). Assuming the example is placed within a package named simple_print, a safer version of Demo would be as follows (the differences are highlighted in bold):

```
public aspect Demo {
   pointcut greetingExecution():
      execution(void greeting())
      && within(simple_print.*);

   before(): greetingExecution() {
      System.out.println("Before the greeting.");
   }
}
```

```
class HelloWorld {
   void greeting() {
      System.out.println("Hello, World!");
   }
   public static void main(String[] args) {
      HelloWorld hello = new HelloWorld();
      hello.greeting();
   }
}
```

```
public aspect Demo {
   pointcut greetingExecution():
      execution(void greeting());

   before(): greetingExecution() {
      System.out.println("Before the greeting.");
   }
}
```

Listing 4: Simple example of pointcuts and advice.

Inter-type Declarations

The purpose of *inter-type declarations* – also known as *static introductions*, or simply *introductions* – is to declare additional fields and methods into classes. For example:

```
   public void Foo.actionPerformed(ActionEvent e) {
      System.out.println(this + " is executing.");
   }
```

This definition adds the method actionPerformed to the target class Foo. From the perspective of the clients of Foo, it is as if the method was declared within the class Foo. Another kind of introduction is to change the inheritance hierarchies (with some limitations) of classes. One frequent use of this mechanism is to make a class implement additional interfaces. It is possible (and very useful) to make classes implement interfaces declared within the aspect.

Inter-type declarations play a very important part in making AspectJ a powerful tool for modularising crosscutting concerns. One contributing factor is the fact that access modifiers on inter-type members are scoped with regard to the aspect that declares them, not the target type. Therefore, when class members declared by the aspect are declared private, they are private to the aspect and the only place in the source code in which those members can be used is the aspect. This rule further enforces the modularity of aspects.

This capability can solve a crosscutting problem that usually has a considerable negative impact whenever arises. When a given set of data items must be passed as arguments throughout several layers of methods, the data must be included in signatures of various methods across layer boundaries, "polluting" the interfaces [91][90]. This causes a scattering and tangling problem in a system that otherwise would comprise a well-modularised structure with several independent layers of abstraction. AspectJ enables those data items to be used exclusively within an aspect, eliminating this interface pollution and yielding a cleaner structure.

Declare Clauses

AspectJ includes a number of 'declare' clauses that configure or extend the behaviour of the compiler, in a way that goes toward the concept of *active libraries* [35]. Though not all such AspectJ clauses are referred in the refactorings documented in Chapter 6, a few of them play an important part in the refactorings. We next mention those clauses.

AspectJ provides 'declare warning' and 'declare error' clauses to instruct the compiler to issue additional warnings and errors respectively. Both clauses associate a pointcut with a warning or an error, and a corresponding message. For instance, the following 'declare error' exposes all the points in which the System.out and System.err objects are accessed within a given scope. The intention is to prevent the program from using printing messages to the console.

```
aspect CatchSystemPrinting {
   pointcut scope(): //some scope definition...
   declare error:
      scope() &&
      (get(* System.out) || get(* System.err) ||
       call(* Throwable.printStackTrace())):
      "Don't print, use the logger";
}
```

Declare warning and 'declare error' clauses are frequently used to enforce system-wide policies like the one above. Not all pointcut designators can be used with these clauses, as they comprise a static (i.e. compile-time) mechanism and some pointcut designators specify joinpoints which can be resolved only during runtime (i.e. they include a *dynamic residue* [64]). Despite these limitations, 'declare warning' and 'declare error' clauses proved to be very useful, and that is demonstrated in several of the refactorings from Chapter 6. The most common uses are to assist the developer in detecting scattered points of interest during a search (i.e. the so-called *scout aspects*) and in policy-enforcement.

The 'declare parents' clause is also widely used but for a very different purpose. With it, aspects can declare a different supertype (e.g. class or interface) of a given class. The new supertype must be a subtype of the original supertype. Therefore, 'declare parents' can make a class implement additional interfaces. The target class must provide the operations declared by the interface (i.e. it must be a subtype of the interface), or the compiler will issue an error. For instance, the following example:

```
   declare parents: BankAccount implements java.io.Serializable;
```

makes the BankAccount class implement the Serializable marker interface from Java's standard API.

In addition, the aspect can also introduce the *concrete* operations using inter-type declarations. This enables AspectJ to emulate one kind of multiple inheritance, mixin inheritance [22]. The combination of inter-type declarations and 'declare parents' opens the way for an interesting idiom. It comprises an abstract aspect using an interface (possibly inner to the aspect) to model an additional role for a set of objects. The aspect declares additional state and behaviour on the interface. Concrete aspects use 'declare parents' clauses in to specify what classes actually play the role in a particular program. That is the pattern used by many of the examples presented in Chapter 4.

2.4.3 A Complete Illustrating Example

We next illustrate some of the capabilities of AspectJ to handle crosscutting concerns with simple example (see Listing 5).

```
public class EuphoricHello {
   static {
      System.out.println("EuphoricHello class loaded");
   }
   public EuphoricHello() {
      System.out.println("Instance of EuphoricHello (" + this + ") created.");
   }
   public void salute() {
      System.out.println(this + ": Oh what a wonderful world!");
   }
}
```

```
public class TragicHello {
   static {
      System.out.println("TragicHello class loaded");
   }
   public TragicHello() {
      System.out.println("Instance of TragicHello created.");
   }
   public void greeting() {
      System.out.println("Goodbye, cruel world!");
   }
}
```

```
public class NervousHello extends TragicHello {
   static {
      System.out.println("NervousHello class loaded");
   }
   public NervousHello() {
      System.out.println("Instance of NervousHello created.");
   }
   public void greeting() {
      EuphoricHello[] euphoria = {new EuphoricHello(), new EuphoricHello(),
            new EuphoricHello(), new EuphoricHello()};
      for (int i = 0; i < euphoria.length; i++ )
         euphoria[i].salute();
      super.greeting();
      System.out.println("We live in a dangerous world!");
   }
}
```

```
public class ComplexHello {
   public static void main(String[] args) {
      (new NervousHello()).greeting();
      System.out.println("Hello, world!");
   }
}
```

Listing 5: Messages to the console as a crosscutting concern

Several classes are included to maximise the crosscutting element as much as possible – each a variant of the celebrated "hello world" example. One of the objects creates an array of other objects. Each class from the example makes a call to the println method of the System.out object in various moments of the class' life cycle – when the class is loaded (the static block), when an instance is created (within the constructor) and when a method is called. The important thing to note is that the entire code base is sprinkled with such calls to the println method. When we run the example by calling the main method of ComplexHello, we see an output like the following (capPrints was the name of the enclosing package):

```
TragicHello class loaded
NervousHello class loaded
Instance of TragicHello created.
Instance of NervousHello created.
EuphoricHello class loaded
Instance of EuphoricHello (capPrints.EuphoricHello@194df86) created.
Instance of EuphoricHello (capPrints.EuphoricHello@defa1a) created.
Instance of EuphoricHello (capPrints.EuphoricHello@f5da06) created.
Instance of EuphoricHello (capPrints.EuphoricHello@bd0108) created.
capPrints.EuphoricHello@194df86: Oh what a wonderful world!
capPrints.EuphoricHello@defa1a: Oh what a wonderful world!
capPrints.EuphoricHello@f5da06: Oh what a wonderful world!
capPrints.EuphoricHello@bd0108: Oh what a wonderful world!
Goodbye, cruel world!
We live in a dangerous world!
Hello, world!
```

Suppose we would like to change the messages the system sends to the console, *without* changing the existing source code. This kind of change is hard, and sometimes impossible to achieve with traditional techniques. To illustrate the capabilities of AOP, suppose the aspect shown in Listing 6 is included in the system's build.

```
01 public aspect BracketEncloser {
02    pointcut printlnToConsole(String message):
03       call(public void java.io.PrintStream.println(String))
04       && !within(BracketEncloser)
05       && args(message);
06
07    void around(String message): printlnToConsole(message) {
08       System.out.println("[" + message + "]");
09    }
10 }
```

Listing 6: Example of a simple aspect enclosing messages in brackets

The BracketEncloser aspect encloses each of the messages sent to the console within square brackets. It does this by intercepting all calls to System.out.println, in the process capturing the string containing the message. The printlnToConsole pointcut (Listing 6, lines 2-5) specifies the relevant calls. It uses the within pointcut designator to prevent capturing the calls to println made within the aspect itself. This is important, because otherwise it would result in infinite recursion. The around advice acting on the pointcut (Listing 6, lines 7-9) overrides the joinpoint with a send of the message enclosed with the brackets. The output sent to the console when the system is built with BracketEncloser aspect is like what follows:

[TragicHello class loaded]
[NervousHello class loaded]
[Instance of TragicHello created.]
[Instance of NervousHello created.]
[EuphoricHello class loaded]
[Instance of EuphoricHello (capPrints.EuphoricHello@1a16869) created.]
[Instance of EuphoricHello (capPrints.EuphoricHello@1cde100) created.]
[Instance of EuphoricHello (capPrints.EuphoricHello@16f0472) created.]
[Instance of EuphoricHello (capPrints.EuphoricHello@18d107f) created.]
[capPrints.EuphoricHello@1a16869: Oh what a wonderful world!]
[capPrints.EuphoricHello@1cde100: Oh what a wonderful world!]
[capPrints.EuphoricHello@16f0472: Oh what a wonderful world!]
[capPrints.EuphoricHello@18d107f: Oh what a wonderful world!]
[Goodbye, cruel world!]
[We live in a dangerous world!]
[Hello, world!]

The BracketEncloser aspect influenced the final behaviour, but in this example the changes are arguably somewhat superficial. Aspects have the power to affect the behaviour of the primary code in fundamental ways. For instance, the aspect shown in Listing 7 sorts all the messages sent to the console in lexicographical order, ignoring case differences. It does this by intercepting the calls to println as before (Listing 7, lines 23-26), but in this case, it inserts the messages in a list, in the appropriate positions (Listing 7, lines 28 and 15-17). The advice and does not call proceed, which means the original behaviour was deleted.

```
01 public aspect OutputReordering {
02    private List _sortedMessages = new Vector();
03
04    private int findInsertionIndex(String message) {
05       int index=0;
06       while(true) {
07          if(index >= _sortedMessages.size()) break;
08          if(message.compareToIgnoreCase((String)_sortedMessages.get(index))
09              <=0)
10             break;
11          index++;
12       }
13       return index;
14    }
15    private void insert(String message) {
16       _sortedMessages.add(findInsertionIndex(message), message);
17    }
18    private void printMessages() {
19       for(Iterator it = _sortedMessages.iterator(); it.hasNext();) {
20          System.out.println(it.next());
21       }
22    }
23    private pointcut printlnToConsole(String message):
24       call(public void java.io.PrintStream.println(String))
25       && !within(OutputReordering)
26       && args(message);
27    void around(String message): printlnToConsole(message) {
28       insert(message);
29    }
30    private pointcut mainExecution():
31       execution(public static void main(String[]));
32    after(): mainExecution() {
33       printMessages();
34    }
35 }
```

Listing 7: Example of an aspect sorting messages to the console

The aspect includes a second pointcut to detect the end of the execution of the method that initiated the system's execution (Listing 7, lines 30-31). An after advice then prints the sorted messages (Listing 7, lines 32-34 and 18-22).

When built with the OutputReordering aspect, the system generates the following output when executed:

```
capPrints.EuphoricHello@16f0472: Oh what a wonderful world!
capPrints.EuphoricHello@18d107f: Oh what a wonderful world!
capPrints.EuphoricHello@1a16869: Oh what a wonderful world!
capPrints.EuphoricHello@1cde100: Oh what a wonderful world!
EuphoricHello class loaded
Goodbye, cruel world!
Hello, world!
Instance of EuphoricHello (capPrints.EuphoricHello@16f0472) created.
Instance of EuphoricHello (capPrints.EuphoricHello@18d107f) created.
Instance of EuphoricHello (capPrints.EuphoricHello@1a16869) created.
Instance of EuphoricHello (capPrints.EuphoricHello@1cde100) created.
Instance of NervousHello created.
Instance of TragicHello created.
NervousHello class loaded
TragicHello class loaded
We live in a dangerous world!
```

2.4.4 Aspect Interfaces and Reusable Aspects

In [55], Gudmundson and Kiczales propose a novel kind of interface, based on pointcuts. They explain that named pointcuts (pointcuts can also be anonymous) are similar to methods in important ways. Named pointcuts have both a signature and a definition, and other constructs can access them by name. To further clarify their semantics, named pointcuts usually include a brief description in the source code/or interface documentation (in Java the former are typically use to generate the latter through the use of tools such as javadoc), the same way as method signatures.

Gudmundson and Kiczales note that the abstraction capability of named pointcuts is similar to that of a method signature. The pointcut has a name and a parameter list, just like a method signature. The signature of a pointcut can convey the abstract operation captured by the pointcut definition, just as the signature of a method conveys the abstract operation performed by the method implementation. These capabilities open the way for a new kind of interface, based on pointcut signatures, and provide the reasoning behind the capability of AspectJ to declare abstract pointcuts in abstract aspects, which are concretised by concrete subaspects. However, it is worth mentioning at this point that AspectJ does not allow for polymorphism, as an aspect can only inherit from another aspect if the latter is abstract.

The reusable aspects described in Chapter 4 provide good examples of the kind of interface that Gudmundson and Kiczales proposed. This is illustrated in Listing 8, which shows the ChainOfResponsibilityProtocol abstract aspect – the reusable implementation of the Chain of Responsibility pattern ([48], p.223) [60]. All header comments were removed for brevity, with the exception of the one for the eventTrigger pointcut. The abstraction captured by eventTrigger is the set of interesting events to which a handler should be updated. Each different case requires its own unique set of joinpoints, and for this reason the eventTrigger pointcut is abstract. Each case-specific aspect dealing with a specific implementation of

Chain of Responsibility must inherit from the abstract aspect and concretise eventTrigger, in a way that is similar to the use of abstract methods in class inheritance hierarchies.

Figure 6 illustrates the rationale behind abstract aspects. Aspect pointcuts enable a component to adapt to the "form" of a code base. The ability of aspects to adapt to different forms directly depends on the joinpoint model supported by the aspect-oriented language. Each different code base has its own unique "form", which comprises all elements that can be "seen" through the joinpoint model. For instance, the joinpoint model of AspectJ covers method names, and therefore a change in the name of a method potentially entails a change to the "form" of the code base, as seen by the aspects.

```
public abstract aspect ChainOfResponsibilityProtocol {
   protected interface Handler {}
   protected interface Request {}

   private WeakHashMap _successors = new WeakHashMap();

   public boolean Handler.acceptRequest(Request request) {
      return false;
   }
   public void Handler.handleRequest(Request request) {}

   /**
    * The join points after which a request is raised.
    * It replaces the normally scattered calls to notify(). To be
    * concretized by sub-aspects.
    */
   protected abstract pointcut eventTrigger(Handler handler, Request request);

   after(Handler handler, Request request): eventTrigger(handler, request) {
      receiveRequest(handler, request);
   }

   public void setSuccessor(Handler handler, Handler successor) {
      _successors.put(handler, successor);
   }
   public Handler getSuccessor(Handler handler) {
      return ((Handler) _successors.get(handler));
   }
   protected void receiveRequest(Handler handler, Request request) {
      if (handler.acceptRequest(request)) {
         handler.handleRequest(request);
      } else {
         Handler successor = getSuccessor(handler);
         if (successor == null) {
            throw new ChainOfResponsibilityException(
                         "request unhandled (end of chain reached)\n");
            // This is one way to deal with unhandled requests.
         } //else {
         receiveRequest(successor, request);
      }
   }
}
```

Listing 8: Reusable AspectJ implementation of *Chain of Responsibility*

A reusable abstract aspect encapsulates the parts that are common to all instances of a crosscutting concern, i.e. the parts that are not dependent on the specific "forms" of specific code bases. The aspect may declare abstract pointcuts which enable it to plug to the code bases, but cannot define the concrete joinpoints because these are different for each case (and are subject to changes as the code base evolves). In order to be plugged to a code base, the abstract aspect must be completed with a concrete subaspect defining the concrete pointcuts that are specific to that code base. We expect that many

implementations of crosscutting concerns developed in the next few years will follow this pattern.

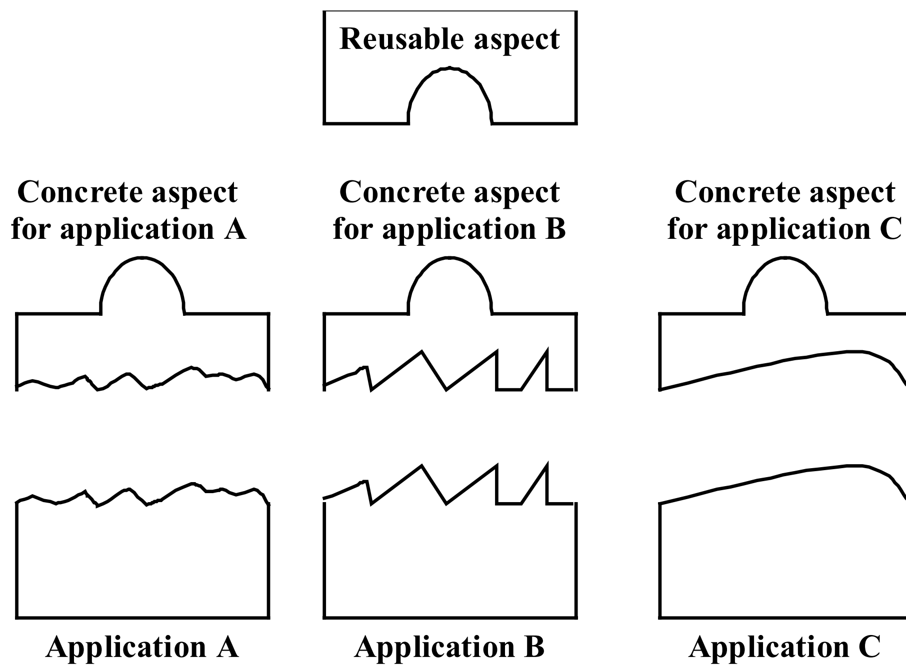

Figure 6: General structure of reusable aspects

Chapter 3. Case Study: WorkSCo

WorkSCo was the first of two case studies used to derive the necessary experience and knowledge to meet the goals stated in Chapter 1. The original aim was to refactor the code of WorkSCo in order to yield an aspect-oriented version of the code. According to the initial plan, the insights gained in the process would be used to develop the collection of refactorings that are the central goal of this thesis. The plan underwent gradual adjustments as the circumstances changed, until the case study was abandoned in favour of the second case study (described in Chapter 4). The present Chapter describes the important details about the case study, including the characteristics of WorkSCo, the experiments performed on it, and the description of hurdles and issues encountered. We believe all these can be useful to practitioners.

Extracting crosscutting concerns from the code base is the natural way to start a refactoring process targeting an OO code base, and the experiments went as far as the first extraction. It also led to the characterisation of the first refactorings of the collection documented in Chapter 6, focusing on the extraction of aspects [118]. These were:

- *Partition Constructor Signature* (137)
- an earlier version of *Move Field from Class to Inter-type* (104)
- an earlier version of *Move Method from Class to Inter-type* (106)
- an earlier version of *Extract Fragment into Advice* (94)
- an earlier version of *Extract Feature into Aspect* (90).

The most significant differences between the earlier versions of the refactorings and those documented in Chapter 6 are in *Extract Feature into Aspect* (90), in which the earlier version does not consider inner classes. In addition, the earlier versions [118] also did not include elaborate motivation sections, but a mere statement of preconditions, when needed.

The experience gained from this case study was valuable in providing a feel of the kind of issues that arise when dealing with non-trivial software systems. That experience was valuable throughout the analysis of the second case study. However, the experiments on WorkSCo were carried out when we did not completely absorb some issues related to style[11], as advocated by Fowler et al. [47]. Consequently, we did not react as aggressively to some style problems found in WorkSCo's code base, as it would later have been the case. Another problem was the fact that the concern that was effectively extracted from the code base did not prove adequate for the AspectJ language, though that conclusion was derived in consequence of the experiment. With hindsight, we believe this extraction can be regarded as the instance of a latent anti-pattern. We elaborate on this issue in section 3.8.

This Chapter is structured as follows. Section 3.1 briefly characterises WorkSCo. Section 3.2 mentions the characteristics of WorkSCo that motivated its use as a case study. After a time, work on WorkSCo was discontinued, and section 3.3 explains why. Section 3.4 presents the approach we took in our work. Section 3.5 describes the architecture of

[11] With hindsight, we can now say that we were at the time yet to completely overcome the *paradigm shift* problem [88] – the difficulties in fully absorbing the concepts of object-oriented programming. Though we were already knowledgeable of all OO concepts at the theoretical level, we did not yet a grasp of the implications these would have on the details of the code at the "micro level". The collection of code smells presented in played a decisive role in putting to rest that problem.

WorkSCo. Section 3.6 describes our exploration of the code base and describes the feature that comprised the target of our single extraction experiment. It also reports on some interesting hurdles found when exploring the code base and from which lessons can be learned. Section 3.7 describes the work we performed on the code base, the most important part of which is the extraction into an aspect of the concern presented in the previous section. Section 3.7 also includes the analysis we made of another possible candidate for extraction. Section 3.8 presents the conclusions we derived from the case study.

3.1. What is WorkSCO?

WorkSCo (Workflow with Separation of Concerns) is an object-oriented framework for workflow management systems [65] under development by the ESW [186] group at INESC-ID, Portugal. WorkSCo is an instance of the micro-workflow architecture developed by Manolescu in his Ph.D. thesis [104]. The proof-of-concept implementation developed in the context of Manolescu's doctoral work was based on Smalltalk, and the description found in [104] is likewise based on the mechanisms of Smalltalk.

WorkSCo is an implementation of micro-workflow based on Java, using some of the "traditional" object-oriented technology surveyed in Chapter 2, namely design patterns and components. WorkSCo does not rely on any of the popular middleware infrastructure for the support of business environments (such as J2EE or .NET). Some of the functionality typically provided by middleware platforms, such as persistence and authentication, is currently being developed as extension modules.

3.2. Why WorkSCo was Selected

WorkSCo was selected as a case study because it is a suitable example of modern software. The domain of workflow software [65] is a demanding one from the software composition point of view, with requirements that include a multiple of crosscutting concerns of various natures, most notably those associated with middleware. WorkSCo can be regarded as an example of the maximum results in terms of separation of concerns that can be obtained through conventional approaches. Thus, WorkSCo held the promise of yielding insights on what is the difference between mainstream techniques and approaches and those of AOP.

Another point in favour of selecting WorkSCo as a case study was the fact that the WorkSCo team was available to provide information and feedback during the phase of learning the framework's structure and domain-specific concepts. In addition, some of the developers of WorkSCo were aware of the developments of AOP and had a grasp of associated concepts, holding the promise of a facilitated exchange of ideas.

3.3. Why Work on WorkSCo was Discontinued

The structure of WorkSCo comprises a kernel and extension modules (for more information see 3.5). When work on this case study started, the functionality kernel was already relatively stable, but the majority of the extension modules were still under development. For this reason, the development of an aspect-oriented version of WorkSCo should be restricted to the kernel in the initial stages. As the modules became ready, they would provide opportunities to assess the compositional capabilities of AspectJ, through the attempt to compose them to the aspect-oriented version of the kernel.

However, while work on a snapshot of the code WorkSCo's kernel was under way, the development team of WorkSCo was facing hurdles related to the composition of the

extension modules to the kernel. Though the kernel's functionality was stable, its code was in a flux, due to the constant need to provide additional hooks enabling the composition of the extension modules. At a certain point, the development team decided to use the AHEAD (Algebraic Hierarchical Equations for Applications Design) tool suite [168] to perform the compositions that would otherwise be troublesome and the code base was adapted accordingly.

Unfortunately, the decision to use AHEAD undermined the value of WorkSCo as a case study for the purposes of this thesis. Using AHEAD meant that Java code would be generated, rather than actually written by the programming team. It is impractical (indeed nonsensical) to apply refactorings to generated code. We could neither continue our work with updated versions of WorkSCo's code base nor benefit from new versions of extension modules that were being developed at the time. In addition, the modifications of the kernel were also undermining the value of the unit tests developed for our snapshot of the kernel (see 3.6.2), which were not being integrated in WorkSCo. Taking a new version of WorkSCo would require an additional effort to adapt or recreate the base of unit tests.

Meanwhile, other interesting candidates for case studies were being considered and we finally decided that a different kind of case study – the one described in Chapter 4 – would be preferable for expanding the collection of refactorings initially derived from the work on WorkSCo.

3.4. Approach Taken

We took the approach of performing refactoring experiments on the WorkSCo code base. The code transformations would be carefully recorded, along with the motivation for performing them. From these sequences of transformations, some patterns would be likely to gradually emerge, and these patterns would provide the basis for the first descriptions of refactorings. These would be analysed in detail to assess whether they were "atomic" enough, or could be further decomposed into smaller steps. The resulting refactorings would then be tested on different code examples.

Extracting a crosscutting concern to an aspect seemed the natural way to start the experiments, as most refactoring processes targeting OO code are likely to start this way, probably followed by further refactorings of the resulting code (both the extracted aspects and the remaining base code). For this reason, the first phase of the work – learning the domain specific concepts – was done with an eye on the crosscutting concerns that would comprise suitable targets for extraction to aspects.

3.5. Architecture of WorkSCo

Manolescu claims that his architecture is the first to specifically target developers of OO software, rather than end-users. The design of micro-workflow relies on a considerable number of design patterns, most of which are intended to achieve particular separations of concerns. The architecture comprises a lightweight kernel providing basic workflow functionalities that are compounded by extension modules offering features that are more advanced. Software developers select the features for their domain-specific workflow processes and add the corresponding modules through composition.

One of the kernel's key abstractions is the concept of *procedure* (Figure 7), which models various kinds of activities commonly performed by workflow systems. The various types of procedures are organised according to the Composite design pattern ([48], pp.163-173). This pattern enables the rest of the code to abstract from the concrete kinds of steps or

activities that comprise the workflow definition, thus allowing the easy extension of the framework with new kinds of procedure. These include both atomic steps (instances of simple procedure) and composite activities (e.g. *composite procedure*). Examples of atomic procedures are *primitive procedure*, which model the actions performed by a single domain-object, and *work list procedure*, which is an abstraction for a piece of work carried out by a human element of the organisation. Composite procedures, namely *sequence procedure*, *conditional procedure* and *repeat-until procedure*, model the conditional and iterative control structures similar to those found in imperative programming languages. They include steps (child procedures) that are also procedures (either simple or composite). Thus a workflow definition has a tree structure, in which all leaf nodes must be instances of the *simple procedure* type (e.g. primitive procedure or work list procedure) and all non-leaf nodes are instances of the composite procedure type.

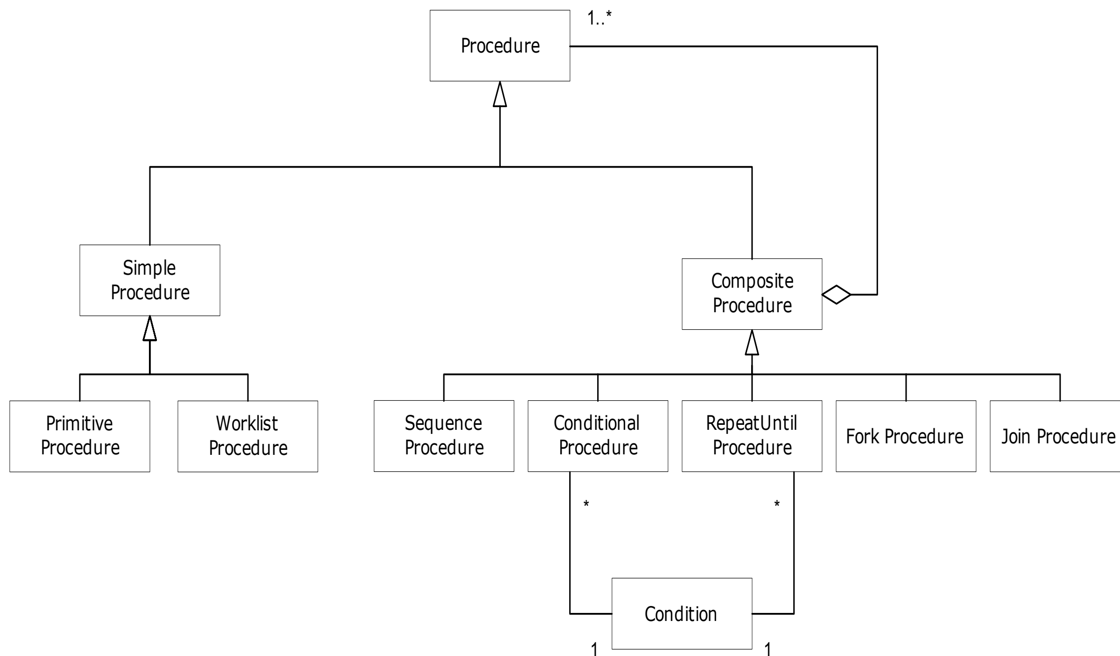

Figure 7: The WorkSCo front-end procedure hierarchy.

The procedure system provides the basic framework for workflow management, enabling domain objects, which typically change less frequently than the glue code that models processes, to be reused as process rules evolve and different process instances are created. Micro-workflow is capable of holding various workflow definitions (representations of workflow processes) using a structure that follows the *Type Object* pattern [71].

The ESW group adapted the original micro-workflow design for their project, which is based on Java technology. In the process, they introduced several structural innovations in order to attain further separations of concerns. The primary design change was motivated by the lack of a standard workflow definition language (WfDL). This made it highly desirable that WorkSCo be able to support several differing, and evolving WfDLs, and to support more than one WfDL simultaneously. The adopted solution was to split the design between two layers – a front-end and a back-end [43] (Figure 8). The back-end layer comprises a *Workflow engine*, which accepts and runs low-level graph representations of the workflow definitions. Each instance of the front-end layer is responsible to handle the concepts and abstractions of a specific WfDL and to compile into the graph structure the back-end understands. The WfDL currently supported by WorkSCo is micro-workflow, which thus became the first instance of the front-end layer.

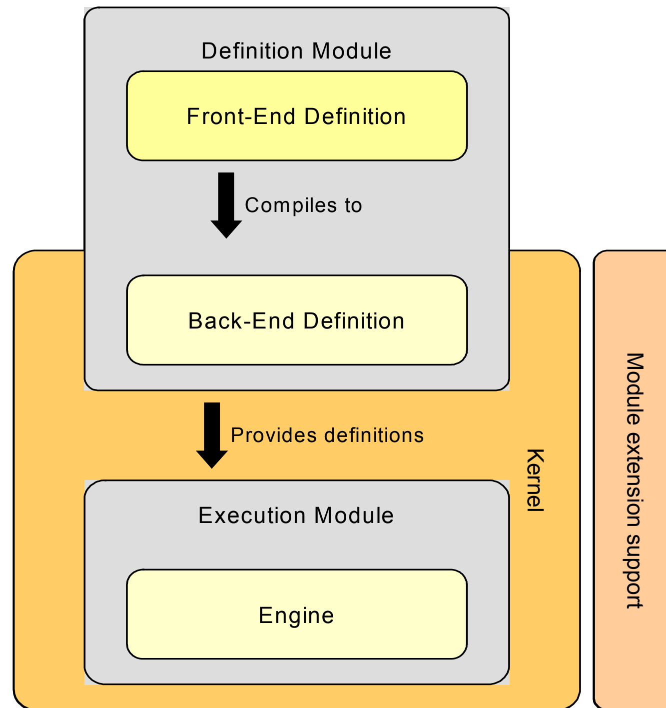

Figure 8: Architecture of the WorkSCo framework

Each front-end instance is responsible for generating the low-level graph structure for the back-end. In the case of micro-workflow, it is done through a traversal of the workflow definition's procedure tree. Generation of the back-end graph is performed bottom-up, starting with the leaf nodes and navigating upwards until the root node is reached. Each node provides a compile operation that generates the graph representation of the sub-tree from which that node is the root. Non-leaf nodes (composite procedures) are responsible to bind the sub-graphs generated by their children, so that the compile operation of root node produces the graph representation of the entire workflow definition. These traversals of the workflow definition's procedure tree effectively comprise an instance of the Interpreter pattern ([48], p. 243-255).

3.6. Exploration of WorkSCo

The kernel functionality was deployed in a package containing 40 classes and interfaces plus a few sub packages providing some accessory functionality. The kernel presented very few dependencies on the code of the extension modules, the only exception being a case in which one of the key classes in the kernel included in its implements clause an interface declared in one of the modules. This low coupling, together with the fact that extracting the data links into an aspect affected only the kernel, enabled us to largely ignore the extension modules and concentrate our analysis on the kernel. Another reason for ignoring the extension modules was the fact that most of them were still under active development and their functionalities still incomplete.

The initial phase of the work consisted in acquiring the necessary grasp of the important domain concepts and gaining a familiarity with the system's structure. During the process we tried to identify crosscutting concerns that would provide suitable targets for extractions. Note that the activity of concern mining was not our focus and for this reason we did not extensively explore sophisticated mining tools and techniques. The learning process mostly relied on eclipse's JDT environment for developing Java applications [179]

to assist in our analysis. JDT's support for code navigation, including the structure view and search support capabilities was very useful during the process of searching the code for units and fragments related to the target concern. We also experimented using the FEAT [138] plug-in for eclipse as a complement to the code search capabilities of JDT. Unfortunately, it proved to be of limited use in our particular case due to FEAT not covering internal details of methods such as local variables. In other words, FEAT does not capture the use relation between components, a widely used relationship in WorkSCo. WorkSCo relies less on static relationships (i.e. inheritance) to connect components than more "traditional" OO frameworks such as the original Smalltalk implementation of micro-workflow described in Manolescu's thesis [104]. These tools were compounded by the feedback of the WorkSCo team, which was helpful in identifying a feature suitable for extraction, as well as gaining enough an understanding of details to start writing unit tests.

We detected a few crosscutting concerns during the exploration process. Unfortunately, the functionality of most of them was not fully developed. They were included by the WorkSCo developers as placeholders for concepts and functionality that, though not necessary in the immediate future, the developers anticipated would be required at a later phase. Since their full development was not a priority at the time, their functionality was left in an incomplete though compilable state. These concerns did not comprise good targets for refactorings because refactorings are meant to preserve the existing behaviour of the code base. Since behaviour is incomplete, it cannot be captured in unit tests and there would be no simple way of knowing whether a given transformation would be illegal if the functionality was complete. This restricted the available choices of targets. Even the functionality of the concern that was selected was not complete, though it was in a sufficiently developed state to be feasible as candidate for refactoring.

3.6.1 The Data Link Feature

Although the front-end model of WorkSCo is control-driven, it enables the independent modelling of data flow between procedures, which is done through *data links*. These provide the necessary framework for communication and data passing among independent domain-nodes. They ensure the independence of the workflow steps and their reusability, and enable the system to provide various services, for example logging and monitoring. The concept of data link belongs to the front-end, while the back-end uses a different, lower level, representation.

Data links are defined at the composite procedure level, so that all control structures have access to it. However, the specific front-end rules must still be enforced, to ensure data link specifications are consistent with the control flow. For instance, data links can only be placed in composite procedures specifying some ordering of their children (e.g. it would not make sense to establish a link between the two branches of a conditional procedure).

The code related to data links was not modularised, being scattered throughout all classes of the procedure hierarchy (Figure 7) starting with CompositeProcedure. This was making it impossible to build a version of WorkSCo devoid of data links, for those clients that do not require such functionality. Therefore, data links seemed a good candidate for extraction into its own module.

The data link code comprised four types of code sections: fields, methods, code fragments in constructors and code fragments in methods. The key element was a field (_dataLinks) declared in the composite procedure class and used by all subclasses. Code using this field could be categorised into the following groups:

- Initialisation code within the constructors of the various composite procedure classes.
- Management methods, including accessors (getters and setters), queries and methods to add, replace and remove a data link. These were all declared in the CompositeProcedure class, and thus inherited by all composite procedures (see Figure 7).
- The compilation logic to provide the back-end graph representation with the low-level equivalent of the data link.

The compilation logic comprised large code fragments placed at the end of the compile method of each composite procedure. These fragments comprised by far the largest parts of the code related to data links. All the compile methods of composite procedures were very large, and the parts related to the compilation of data links were actually larger than the remaining part of those methods. This is illustrated in Listing 9, which shows one of these compile methods (the part related to data links is shaded).

3.6.2 Initial Hurdles

Several issues had to be solved before we felt ready to start refactoring the code. The most serious was the lack of unit test coverage. The WorkSCo project included a collection of various broader-scoped tests that fed the system with example workflows. Unfortunately, these were unsuitable for guaranteeing preservation of behaviour in refactoring processes, due to their coarse grain. As pointed out in [47], fine-grained unit tests are crucial to ensure the code transformations always preserve the original behaviour. From our experience, we can attest the vital importance of unit tests to give what Beck called "courage" in changing the existing code [12]. Only after the initial effort of writing tests did we acquire the confidence necessary to start refactoring the code. Before we reached that point, we simply felt "paralysed with fear". In addition, the act of writing tests greatly helped to make one understand the workings of the system[12] (as suggested by various authors, namely Manolescu [105]). The unit tests created for WorkSCo focused mainly on the compile operations of the various procedures.

Another hurdle was the fact that the elements of both the front-end and the back-end were still bundled together in the kernel package. This included all the components represented in Figure 8, with the exception of the extension modules at the right of the figure. This resulted in a tangling effect at the package level that had some impact on the initial exploration stage.

Another issue was choosing the right approach to deal with possible continuous changes in the code base, which was the subject of continuing development. We were not part of WorkSCo's development team and its code could be changed at any time: whatever version we would operate on, it could quickly become outdated. Our main target was the kernel package of WorkSCo, whose functionality was stable. Unfortunately, functional stability did not preclude modifications on the kernel's code, either to correct defects (i.e. *bugs*) or to change its structure in order to make it more amenable to the composition of the extension modules that were in various stages of development. This entailed providing the kernel's code with whatever structural *hooks* would be needed to enable the required compositions. For the aforementioned reasons, we decided to "freeze" the code base for some time, by taking a snapshot of WorkSCo and undertaking our experiments on it, thus temporarily disregarding the subsequent modifications operated by the WorkSCo development team.

[12] The process of writing unit tests also helped to expose a few bugs in the system.

```
public Graph compile() {
   GraphNode begin = new GraphNode(
      getId() + "B", "Fork", new IdentityService(),_input, _input);
   GraphNode end =
      new GraphNode(getId() + "E", "Fork", new IdentityService(),
                     _output, _output);
   Graph result = new Graph(begin, end);
   if (getChildrenCount() == 0)
      result.addControlFlow(begin, end);
   else {
      List compiledBranches = new ArrayList(getChildrenCount());
      for (int i = 0; i < getChildrenCount(); i++) {
         Procedure branch = getChild(i);
         Graph branchGraph = branch.compile();
         compiledBranches.add(branchGraph);
         GraphNode branchStart = branchGraph.getStart();
         GraphNode branchEnd = branchGraph.getEnd();
         result.addGraph(branchGraph);
         result.addControlFlow(begin, branchStart);
         result.addControlFlow(branchEnd, end);
      }
      // process the data links now that all nodes are in the resulting graph
      for (int i = 0; i < _dataLinks.size(); i++) {
         DataLink dataLink = (DataLink)_dataLinks.get(i);
         String sourceProcedureId = dataLink.getSourceProcedureId();
         String targetProcedureId = dataLink.getTargetProcedureId();
         if(sourceProcedureId.equals(_id)){//this procedure is the source of the
data link
            /* assuming the default behaviour for mapping then the target
             * is the first node of the graph for the target branch. */
            // find index of procedure with id == targetProcedureId
            int index =
               getChildren().indexOf(
                  ProcedureUtils.searchChildProcedure(this, targetProcedureId));
            // get the corresponding graph
            Graph targetGraph = (Graph)compiledBranches.get(index);
            // get the start node, which is the target node
            GraphNode targetNode = targetGraph.getStart();
            //create a data flow between the start node of 'this' graph and the
targetNode
            result.addDataFlow(dataLink.getName(),
                                 begin, targetNode, dataLink.getMaps());
         } else
            // this procedure is the target of the data link
            if (targetProcedureId.equals(_id)) {
               /* assuming the default behaviour for mapping then the source is
                * the last node of the graph for the source branch. */
               // find index of procedure with id == sourceProcedureId
               int index =
                  getChildren().indexOf(
                     ProcedureUtils.searchChildProcedure(this, sourceProcedureId));
               // get the corresponding graph
               Graph sourceGraph = (Graph)compiledBranches.get(index);
               // get the end node, which is the source node
               GraphNode sourceNode = sourceGraph.getEnd();
               // create a data flow between the sourceNode and the end node of
'this' graph
               result.addDataFlow(dataLink.getName(), sourceNode, end,
                                    dataLink.getMaps());
         } else // the data link is between two branch procedures. THIS CANNOT
HAPPEN
            System.err.println("ERROR: Link between branches in ForkProcedure!!!");
      }
   }
   return result;
}
```

Listing 9: Snapshot of the compile method for the ForkProcedure class.

In [135], Pree proposes the concept of *hot spots* of an application framework – "those aspects of the application domain that have to be kept flexible" – and states that "we consider a framework to have the quality attribute "well-designed" if it provides adequate hot spots for adaptations" ([135], p.106). As we point out in section 1.1, this is a hard thing to do upfront in practice, and it was also the case with WorkSCo. These difficulties are also a symptom of the limitations in the composition capabilities of object-oriented techniques (cf. section 1.1).

3.6.3 Style Issues

Though generally well structured, the snapshot of the code we worked on placed a few hurdles. One was the fact that the code did not adhere to the style advocated by Fowler et al. [47]. Style problems were most noticeable in:

- Very large methods responsible for generating the back-end graph representation of workflows. This is illustrated in Listing 9[13], which shows the compile method for one of the composite procedure classes. This method is much smaller than the largest of all compile methods, which had 110 lines of code. Some unrelated methods were even larger. This style problem is characterised by the *Long Method* smell ([47], p.76).
- Long parameter lists, mostly in constructors, e.g. the *Long Parameter List* smell ([47], p.78). One example is illustrated in Listing 10, which shows the constructor of the RepeatUntilProcedure class.
- Many uses of comments explaining the purpose of the code, e.g. the *Comments* smell ([47], p.87). According to Beck and Fowler, comments are not really a smell, but are often used as a deodorant to bad code. The need of a comment to explain what a block of code does is a sign that the code does not reveal its intention the way it should. Such blocks can be improved, namely by using *Extract Method* ([47], p.110]).
- Complex conditional logic, including (1) the unnecessary use of else legs in conditional control structures[14] and (2) the frequent use of tests for null. Tests for null usually suggest that a more aggressive use of the Null Object pattern [163] would be beneficial, though only an analysis of the specific situation can confirm this assumption. In Listing 11 we show a method with both variations of the smell. Fowler et al. [47] did not propose a specific smell for these symptoms, though they dedicated a whole Chapter to the simplification of conditional expressions, including the introduction of null objects, using *Introduce Null Object* ([47], p.260). Kerievsky proposed the *Conditional Complexity* smell ([76], p.41) and proposed his own version of *Introduce Null Object* ([76], p.301).

```
public RepeatUntilProcedure(String id, String name,
      Precondition precondition,
      IOMessage input, IOMessage output,
      ArrayList dataLinks,
      Condition guard, Procedure body) {
   super(id, name, precondition, input, output, dataLinks);
   _guard = guard;
   addChild(body);
}
```

Listing 10: Snapshot of the RepeatUntilProcedure class' constructor.

---

[13] Some indentation adjustements were made for the purposes of representing the code in these pages.

[14] This style problem was automatically detected by eclipse.

```
public String getActualValue(ExecutionContext executionContext) {
   if (_value != null) {
      DPrint.println("dkernel", "Actual value is " + _value);
      return _value;
   }
   else {
      DPrint.println("dkernel", "Value is a message. Getting part named=" +
                     _messagePartName.toString());
      Object messagePartValue = executionContext.getInputMessagePart(_messagePartName);
      /* In order to support shared context message parts we need to check if the
       * messagePartValue is null. In that case we get the data from the sharedContext*/
      if (messagePartValue == null) {
         DPrint.println("dkernel", "Getting message part from shared context.");
         messagePartValue = executionContext.getContext().get(_messagePartName);
      }
      DPrint.println("dkernel", "Trying to print the object=" + messagePartValue);
      if (messagePartValue != null)
         return messagePartValue.toString();
      else { // maybe this is not supposed to happen
         DPrint.println("dkernel", "Message part was null");
         return "";
      }
   }
}
```

Listing 11: Snapshot of a method betraying conditional complexity.

3.7. Work on the Case Study

After an initial learning phase, the bulk of our work on WorkSCo comprised the extraction of a crosscutting concern from the code base, which is described next. We also analysed the code with a view to another concern, and we present our findings in subsection 3.7.2.

3.7.1 Extracting the Data Link Feature

We decided to preserve the existing interface of WorkSCo's kernel during refactoring. This way the refactorings would have no impact on the remaining components of WorkSCo, relieving us from taking into account the impact of the refactorings on the extension modules. This situation is similar to the one of *published interfaces*, i.e., a software component that is used by clients, which therefore become dependent of its interfaces. In both cases, the developers of the evolving component cannot change the interface because that would risk breaking client code. Thus we followed the strategy of first modifying the internal structure of a component, as one stage of a larger refactoring, leaving changes to the interface to later stages.

After creating an empty aspect we dealt with each of the implementation elements in turn – fields first, next related methods, next code fragments using the fields – as we prescribe[15] in *Extract Feature Into Aspect* (90). Moving fields and methods was straightforward and done according to *Move Field From Class To Inter-type* (104) and *Move Method From Class To Inter-type* (106).

Dealing with Parallel Constructor Chains

The constructors placed a more complex and interesting problem. Each subclass in the procedure hierarchy adds new arguments in its respective constructors and therefore the parameter lists of the constructors keep getting longer as we go down the inheritance chain. Each constructor makes a super call passing the arguments defined in the constructor of the superclass and next deals with the initialisation of the data specific to its class. Figure 9

[15] The aspect-oriented refactorings mentioned in this section were in the process of being described and refined at this stage. We refer to the code transformations using the names of the resulting refactorings to facilitate the presentation.

shows three of the constructors, with the code related to data links highlighted in bold. Many of the arguments relate to concerns other than the primary concern, data links being one example. Naturally, we wished to place all initialisation code related to the data link concern within the aspect and to keep all other code in the classes. To complicate things, the constructor received arguments related to the concern we wanted to extract but we were constrained by the need to maintain its signature in order not to break existing client code outside the kernel package.

```
public abstract class Procedure implements Cloneable {
   Procedure(String id,
             String name,
             Precondition precondition,
             IOMessage input, IOMessage output) {
        //initialisation code
      }
   //rest of Procedure code
}
```

```
public abstract class CompositeProcedure extends Procedure {
   protected List _dataLinks = null;
   CompositeProcedure(String id,
                      String name,
                      Precondition precondition,
                      IOMessage input, IOMessage output,
                      ArrayList dataLinks) {
      super(id, name, precondition, input, output);
      _dataLinks = dataLinks;
   }
   //rest of CompositeProcedure code
}
```

```
public class SequenceProcedure extends CompositeProcedure {
   public SequenceProcedure(String id,
                            String name,
                            Precondition precondition,
                            IOMessage input, IOMessage output,
                            ArrayList dataLinks,
                            List steps) {
      super(id, name, precondition, input, output, dataLinks);
      //initialisation code
   }
   //rest of CompositeProcedure code
}
```

Figure 9: Original chain of procedure constructors in WorkSCo.

We devised *Partition Constructor Signature* (137) to solve this problem. The intended result was to provide the base code of each composite procedure with a new constructor with a shorter argument list, devoid of the argument related to the extracted concern, and to make the aspect introduce a constructor with the original and longer argument list. The introduced constructor would call the simplified constructor in the class, thus avoiding having code unrelated to the data link concern. This way the original constructor signatures would be preserved, for the sake of the existing code that depended on them, but clients that do not require functionality would then be free to use a simpler, cleaner version of the constructors. This is shown in Figure 10, again with code related to data links highlighted in bold.

However, when we tried to use *Partition Constructor Signature* (137) we realised that we were not considering super calls. These calls were needed to ensure initialisation code in the superclasses would receive their arguments, as well as avoiding duplication of initialisation code such as the one highlighted in Figure 10. Yet, we could not include calls to super because we were already using calls to this, to link the constructors introduced by the aspect to the simpler versions that remained in base code hierarchy. As it stood, either the

introduced constructors would make the super calls as in Figure 9, in which case code related to the this calls would have to be duplicated, or the this calls could be made but the calls to super could not be made and the related code would have to be duplicated. Note that the trade-off between calls to super and to this can be reversed. For instance, if we chose to place a call to super in the introduced SequenceProcedure constructor, instead of a call to this, we would obtain the code as in Listing 12.

Declared in the base code

```
public abstract class Procedure
     implements Cloneable {
   protected String _id = null;
   protected String _name = null;
   protected Precondition
      _precondition = null;
   private List _children = null;
   protected IOMessage _input = null;
   protected IOMessage _output = null;

   Procedure(String id,
             String name,
             Precondition precondition,
             IOMessage input,
             IOMessage output) {
      _id = id;
      _name = name;
      _precondition = precondition;
      _input = input;
      _output = output;
      setEmptyChildren();
      setEmptyProperties();
   }
```

Declared in the DataLink aspect

```
private List
   Procedure._dataLinks = null;

public Procedure.new(
         String id,
         String name,
         Precondition precondition,
         IOMessage input,
         IOMessage output,
         ArrayList dataLinks) {
   this(id,
         name,
         precondition,
         input, output);
   _dataLinks = dataLinks;
   }
```

Declared in the base code

```
public class SequenceProcedure
     extends Procedure {

   public SequenceProcedure(
        String id,
        String name,
        Precondition precondition,
        IOMessage input,
        IOMessage output,
        List steps) {
      super(id, name,
           precondition,
           input, output);
      if(steps==null);
        //TO DO: throw exception
      setChildren(steps);
   }
```

Declared in the DataLink aspect

```
public SequenceProcedure.new(
      String id,
      String name,
      Precondition precondition,
      IOMessage input,
      IOMessage output,
      ArrayList dataLinks,
      List steps) {
   this(id,
         name,
         precondition,
         input, output,
         steps);
   _dataLinks = dataLinks;
}
```

Figure 10: Constructor Hierarchies with and without Data Links

```
public SequenceProcedure.new(String id,
                             String name,
                             Precondition precondition,
                             IOMessage input,
                           ..IOMessage output,
                           ..ArrayList dataLinks,
                           ..List steps) {
   super(id, name, precondition, input, output, datalinks);
   if (steps == null); //TO DO: throw exception
   setChildren(steps);
}
```

Listing 12: Variant of introduced constructor with call to super.

We concluded that *Partition Constructor Signature* (137) required, as a precondition, that the constructors do not pass arguments to calls to super. Therefore, it could not be used for the particular case of the procedure hierarchies. We solved the problem by keeping the calls to this in the introduced constructors, just as in Figure 10, and treating the code related to super as crosscutting code *within* the aspect, which was extracted to its own advice (Figure 11).

Figure 11: Dealing with a crosscutting concern internal to an aspect.

This problem interestingly exposes the existence of two dimensions: (1) the normal inheritance dimension, represented by the constructor chain placed in the primary code and (2) a separate dimension represented by the aspect. There are two separate constructor hierarchies, one of which is introduced by the aspect. The problems make themselves felt when we try to link an element from one hierarchy to an element from another. In such situations are we trying to cross the "aspect dimension".

Another interesting characteristic of the advice shown in Figure 11 is the fact that it applies only the constructors introduced by the aspect itself, i.e. the advice crosscuts only its enclosing aspect. This is an effect similar to those that can be obtained when an inner aspect placed is used within a class, in order to address a crosscutting concern that is restricted to only that class.

The Constructor Explosion Problem

The parameter lists of the constructors of the procedure classes included other arguments associated to other secondary concerns, which like the data links could conceivably be extracted to their own aspects. This brought the question of what happens when we are in the presence of not one, but several aspects, each with its own extra set of parameters to add to the class's constructor signature.

Each of the aspects may export a partial constructor signature related to its respective concern, but clients that require the complete constructor signature that results from the combined functionality won't find it anywhere. AspectJ does not provide a mechanism that directly augments the signature of methods or constructors.

A possible solution would be the creation of a *manager aspect* that deals with issues related to the composition of several concerns. Such an aspect can declare and export the missing compound signature. However, for each different combination of more than one of these aspects, a different constructor signature will have to be implemented, resulting in a combinatorial explosion of constructor signatures.

Capturing Needed State in Advice

Extracting code fragments related to data links from the large compile methods also presented some interesting problems. It is worth pointing out that at this point we found it useful to use[16] *Extract Method* ([47], p.110]), not just to ease the way for subsequent refactorings but primarily to making it easier to reason with, by isolating it from the code related to the primary concern.

The problems were caused by difficulties in capturing the necessary context. Each fragment related to data links used several data structures that were created in the part of the compile method that preceded it. Those structures were referenced by local variables and neither stored as fields of some object nor passed to the compile method as arguments. This caused the problem of how to capture them for an advice within the aspect, since AspectJ's pointcut protocol does not include local variables. Sometimes this kind of problem can be solved by capturing the result value of a method call made within the method of interest. However, we could not do even that, because there was more than one object involved.

It would be *theoretically* possible to capture all the necessary objects, but that would require extremely complex pointcuts, resulting in code hard to understand and probably error prone. We could obtain the needed state through accessors some of the objects after the end of the execution, but at least one complex structure would have to be computed a second time, leading to some code duplication (at least performance is not an issue in workflow applications).

An obvious solution would be to turn the needed local state into fields of the procedure objects, as that would make it very simple to capture them within the aspect. However, that would have been extremely crude and inadequate. We found a compromise by turning the compile methods into method objects, by using *Replace Method with Method Object* ([47], p.135). The method objects enabled us to expose the needed state to the pointcuts but the associated fields would exist only for the duration of the execution of the compile operation, rather than for the entire lifetime the procedure objects. The classes of the method object were created as inner classes within each procedure class. After resorting to this solution, it was straightforward to apply[17] *Extract Fragment into Advice* (94).

A word of caution must be given in respect to this solution. We had to make the fields of the method objects public, so that code within the advice could have access to its fields. This is contrary to common practice in object programming. Otherwise, we would need to provide the method objects with accessor methods, which we felt would be overkill in these circumstances. Another solution would be to make the aspect privileged; something we think should be avoided except as a last resort. Use of privileged aspects is akin to making everything public (for the aspect). As Colyer and Clement remark in [31], aspect privilege confers the general privilege to access any private state anywhere, while one usually wishes to express privilege with respect to a single class or a restricted set of classes.

[16] Throughout the exploration of these code bases we used the eclipse IDE [179]with the AJDT plug-in [169]. AJDT reuses part of the functionality of eclipse's JDT plug-in for Java.

[17] The use of *Replace Method with Method Object* ([47], p.135) is sugested in the motivation section of *Extract Fragment into Advice* (94).

Issues of Data Access

At the start of the refactoring experiment, we created the aspect in a subpackage of WorkSCo's kernel package rather than in the kernel package itself. This would make it easier to configure alternative builds, with and without the additional data link functionality. In addition, we regarded the data links as an accessory functionality, and to place it in a subpackage would make that explicit. However, during the process we started to get compiler errors due to visibility violations – several methods and fields referred from code moved to the aspect had restricted access (protected, private or package). We temporarily solved these problems by (reluctantly) classifying the aspect as privileged, until we analysed the various issues and discovered that none of the methods was private – they had either package or protected access modes, meaning they were indeed supposed to be visible, but only within a restricted scope. In the end we kept the aspect in its own subpackage and solved the problems by refactoring the base code, by encapsulating a few fields accessed in the advice code, using *Self Encapsulate Field* ([47], p.171), and relaxing access clauses of some methods called from the aspect's advice. A class from the kernel was used only by the aspect and it was moved to the data link subpackage.

The resulting Aspect

The aspect that resulted from the extraction process comprised a set of inter-type declarations and a set of pointcuts with the corresponding (mostly after) advice. The inter-type declarations could be grouped into (1) one inter-type field related to the concern, (2) several auxiliary methods to manage the data that the field is holding (access, insertion, removal, query), (3) the alternative versions of the constructors and (3) methods for the actions relating to the compilation of the data link parts. The latter were called from after advice matching the execution of the compile operations. One think to note is that there is no crosscutting effect on the part of the aspect: each pointcut captured only one joinpoint and each inter-type declaration targeted only one class.

The bulk of the code relating to data links is the compilation part extracted from the compile methods of the procedure hierarchy. This hierarchy is a composite structure, and the compilation process comprises the traversal of this structure. The compilation process is therefore an instance of the Interpreter pattern [48], which invites a comparison with the AspectJ implementation of Interpreter presented by Hannemann and Kiczales [60]. Unlike several of the examples from the collection from [60], the implementation of Interpreter comprises a single concrete aspect that does not extend a reusable abstract aspect (i.e. there is no reusable code). The aspect does not include pointcuts and advice, only inter-type declarations. An interesting comment can be found at the beginning of the source file:

"The very nature of the interpreter pattern introduces coupling between all participants. Unfortunately, removing the pattern code from the participants does not work as nicely here as with other patterns. The reason is that the roles are defining, i.e., each participant's functionality is determined (only) by its role. If aspects were used to implement the entire pattern functionality, it would leave the Expressions etc. empty and would make the aspect a monolithic module.

However, it is still possible to augment or change the behaviour of the system without changing all participant classes. To show this, we assumed that BooleanExpression.replace(String, BooleanExpression) and BooleanExpression.copy were added later. An aspect is used to implement those methods, so that other classes do not have to change (we only changed the interface, but even that was not necessary).

In general, however, this pattern does not lend itself nicely to aspectification."

Similar remarks can be made in relation to the data link concern described in this Chapter. The data link aspect is more complex than the interpreter aspect from [60] (e.g. it includes pointcuts and advice) only because the target code base is more complex. In both cases, the aspect is an unyielding structure whose members affect a single point of the primary code each (i.e. there is no crosscutting).

3.7.2 Analysis of the Debugging Concern

Debugging – in this context, writing messages to the console to enable the developer to monitor executing sessions of the application – is a typical crosscutting concern, whose instructions can potentially be placed almost anywhere in the code base. Traditionally, the only way to obtain debug messages in key points of the execution flow is to resort to direct interventions in the source code. The concern is invasively plugged into the code base through insertion of the related instructions in the appropriate points, and if the messages were no longer required, the concern would have to be invasively unplugged by deleting those instructions. As an alternative, the debug concern can be unplugged by placing the instructions within comments. However, this does not eliminate the cluttering effect, and whenever the developer wants to reactivate the messages, it must edit the code yet again.

WorkSCo included a mechanism to ameliorate this problem. It included a class – DPrint – for sending debug messages in a controlled way. DPrint uses a println operation similar to System.out.println, except that it receives as a first argument a string representing a label. DPrint only prints the message (to the console) if the corresponding label was previously registered through a debugOn operation passing that label as argument. All messages associated to a given label can be deactivated by unregistering the label. This is done by calling the debugOff operation with the label as argument. This way it is simple to activate and deactivate sets of calls to DPrint.println, by calling debugOn and debugOff at the appropriate points. The calls to DPrint.println are scattered throughout all the code of WorkSCo.

We considered extracting this debug concern to an aspect. When we analysed the scattered calls to DPrint.println, we found out that it would not be possible to extract it while preserving the externally observable behaviour, exactly as it stood. The reason was that the calls to DPrint.println were placed in the middle of methods, often within control structures. These locations are not supported by AspectJ's joinpoint model. In some cases, the messages included values of local variables, which cannot be captured by the pointcuts of AspectJ. However, we also noticed that if the code were transformed according to the style advocated by Fowler et al. [47], namely by decomposing many large methods, it should be much easier for an aspect to plug a set of calls to DPrint.println close to the one found in the snapshot. The option of using an aspect to plug the calls would likely not be as flexible (in terms of the exact locations of the calls) as placing the calls directly in the intended locations, even after an aggressive refactoring. However, in such case the concern would be modularised, with all the associated benefits.

3.8. Discussion

This section presents some lessons learned from the experiments described in this Chapter. It should be noted that although the case study provided experience and various insights, the work on the case study itself benefited little from those insights. This was so because it was abandoned in preference of the second case study, at a time when the insights and conclusions were still maturing.

3.8.1 Good Object-Oriented Style is a Precondition for applying AOP

The case study described in this Chapter demonstrated that a sound structure of the code, in accordance to the one advocated by Fowler et al. [47], is an essential prerequisite for aspect-oriented refactorings to be applied effectively. Well-formed code bases tend to be richer in joinpoints than badly formed code bases, and greater joinpoint richness is likely to provide aspects with the leverage they require for their quantifications. Sparseness of joinpoints results from a poorly or insufficiently decomposed system. The most serious symptoms of insufficient decomposition were very long methods, large classes and too many classes in the same package.

It is a fortunate coincidence that the style suitable for aspects closely corresponds to the one advocated by Fowler et al. [47]. Thus, when developers consider using aspects on an existing OO code base, they should first ensure that it is well formed in terms of a good OO style, and refactor the code if it is not the case [57].

Making the code of WorkSCo conform with [47] would require, at the very least, an aggressive decomposition of many methods – using refactorings such as *Compose Method* ([76], p.123) and *Extract Method* ([47], p.110). Such refactorings would enrich the joinpoint structure of the base code to a level of granularity sufficient for the extraction of the debugging concern to be feasible, or at least to be possible to create an aspect yielding behaviour close to the original. It would also likely enable developers to extract the code related to data links from the compile operations without the need to use *Replace Method with Method Object* ([47], p.135). This refactoring may be useful in some situations, but if the structure of the code base is sound, these are not likely to arise often. In the case of a refactored WorkSCo, we believe the various objects required for the data link phase would have been captured using new joinpoints resulting from the refactoring. For instance, the intermediate graph structure could conceivably have been captured from the return value or argument of a new method called within a decomposed compile method. Likewise, the constructors of the procedure classes would benefit of a different approach to instance construction that removed some of the constructor arguments, particularly those related to features that are used only in some configurations.

It is likely that if the debugging concern is modularised, the resulting aspect will not provide exactly the same behaviour as the scattered calls to DPrint.println, even after a thorough refactoring that aggressively decomposed many methods. Some of the calls would still conceivably not find a suitable joinpoint. Nevertheless, it is likely that the joinpoints offered would be fine enough to satisfy developers when carrying out a debug session.

Note that the act of refactoring the code base can also be beneficial in itself [105]. Independently of the compositions made possible, a well-refactored code base is easier to understand and to reason with. Applying refactorings on source code, for instance by simply renaming a method or class whose name does not seem to transmit its intent clearly, can provide many useful insights. The same applies to the act of writing unit tests (in case the system does not already have them). As Manolescu states in [105], "It's hard to think of a better way of understanding something than actually doing it!"

3.8.2 Preservation of Intent vs. Preservation of Behaviour

Some of the situations described above raise the issue of preservation of intent versus preservation of behaviour. There are cases when refactoring to aspects is not behaviour preserving. This does not necessarily mean that the intent of the developer is

compromised. In many situations, the intentions of the programmer can be met by many slightly different variants of behaviour.

When discussing a tool for aspect-oriented refactoring, Hannemann et al. in [59] describe the case of refactoring one implementation of the Observer pattern ([48], p.293) in the JHotDraw framework [182]. In the connection between the DrawingView and FigureSelectionListener types, the code notifies listeners after calls to addSelection(Figure), but not after addSelectionAll(Figure[]). The latter method relies on calls to the former method for its implementation. The extracted aspect relied on a straightforward implementation based on pointcuts and advice, which would trigger updates after both kinds of calls. The result would be an additional call to listeners to update after adding an array of figures to the selection. Although a user may not notice the difference, such a refactoring would not preserve the original behaviour. However, the aspect-oriented version would be safer because it captures the intent of the program: listeners should be informed of the addition of figures and if in the future a developer changed the implementation of addSelectionAll(Figure[]) to not using calls to addSelection(Figure), the OO version would no longer inform the listeners. The aspect-oriented version captures the intent through the direct application of the language mechanisms and therefore provides a stronger guarantee of consistency and safety.

Another example of a mismatch between preservation of intent and preserving the actual behaviour (also involving implementations of Observer) is described in Chapter 7. This case relates to the order in which listeners are notified, which changed during the refactoring process. This detail is irrelevant for the purposes of the desirable behaviour and therefore the test that signalled the change in behaviour was changed accordingly.

Both the aforementioned cases are examples of situations in which preservation of intent is not the same as preservation of the existing behaviour. This kind of situation is likely to occasionally arise in aspect-oriented refactorings [61], because these entail replacing behaviour obtained through manual (i.e. error prone) interventions in the code with solutions provided by more reliable language mechanisms. When differences arise, they really expose inconsistencies in the manual solution. In such cases, preservation of behaviour may not be possible and may even be undesirable.

Mismatches between intent and actual behaviour are more likely to occur in large and complex systems because these are maintained by several people. Multiple developers are more likely to fail to enforce a global practice consistently (e.g. omissions, deviations from a policy, and bugs). In most cases where a mismatch is exposed, preservation of intent is more important than preservation of the original, possibly faulty, behaviour.

3.8.3 Feature Decomposition as a Likely Anti-Pattern

The problems related to the extraction of the data link concern suggest that not all crosscutting concerns are equally amenable to be extracted to aspects. This is independent of the style problems of the WorkSCo code base described in section 3.6.3. The extraction of the data link concern was really an attempt to decompose the system according to *use cases* [67] or *features*[18] [74]. The mechanisms AspectJ, particularly inter-type declarations, were used to connect the elements of the aspect to the relevant points in the primary system.

[18] The term “feature” is used with different menanings, depending on the context. When considering criteria for the decomposition of software systems, we use the term “feature” in the sense of [74]. For the purposes of our analysis, we regard the use case and feature decompositions as equivalent.

We conclude from the analysis of this case study that a language like AspectJ is not suitable to decompose a system according to use cases or features (i.e. to support product lines). The reason is that the base decomposition of AspectJ is fixed, and is the same as the decomposition of the language that AspectJ extends – classes and objects. Trying to decompose a system according to a different decomposition is bound to lead to inappropriate and awkward structures. That is root cause of the hurdles described above, regarding our attempt to extract the data links concern (feature).

We propose a simple rule to tell when a crosscutting concern is suitable for extraction into an aspect as understood by languages such as AspectJ: if there is duplication in the scattered code fragments in multiple points of the system, extracting such fragments to an aspect is likely to prove beneficial. Instead of having multiple duplicated code fragments or method calls, the refactored system will include an aspect affecting the appropriate points with one or a few advice. In such a situation, the aspect will yield a system with the same behaviour but less code.

On the other hand, if each code block or fragment related to the concern is different from the others blocks or fragments, extracting these into an aspect is likely to yield a monolithic module with complex connections to the primary structure. The problem can be detected by examining the pointcuts, in case the aspect uses any. If each pointcut captures one joinpoint only, the concern in question is not crosscutting in the sense understood in the context of AspectJ. The same rule applies to inter-type declarations: in general, they should target a set of types rather than a specific type. A system structured with such aspects is not likely to derive benefits from AOP and may become harder to maintain and evolve.

Chapter 4. Case Study: Implementations of the GoF Patterns in Java and AspectJ

Chapter 4 presents the second case study, a collection of code examples comprising the implementation in Java and AspectJ of the GoF patterns. This Chapter explains why the examples were selected (section 4.1) and how they were approached (section 4.2). Next, the Chapter provides some background on the examples (section 4.3), and describes various issues that were detected during analysis and work on them. The Chapter refers to the results obtained (section 4.4) and concludes with a summary of the main insights derived from the case study (section 4.5).

4.1. Why the Examples were Selected

No example implementation of the GoF patterns can be considered "real code". Such code examples must inevitably be labelled "toy code" or something similar. Toy examples cannot be directly used in real applications and are not useful to solve real problems. Toy examples do not generally include additional concerns that place constraints on the design and therefore on the refactorings that we would like to perform in each specific case. Toy examples do not address issues that only arise in complex systems, such as user interfaces, exception handling and the interaction of non-orthogonal concerns.

Nevertheless, the patterns illustrate a variety of design and structural issues, as well as situations that would be hard to find in a single code base (except possibly in some large and complex ones). The implementations of the GoF patterns effectively comprise a microcosm of many possible systems. These examples provide a rich source of insights, without the need to analyse large code bases or learn domain-specific concepts.

At the time we were carrying out our work on the first case study, we were also searching for other suitable code bases. In particular, we were looking for code bases that were publicly available in both their original Java-based form, and in a refactored, AspectJ-based form. There seemed to be none. At the time, the code examples presented by Hannemann and Kiczales in [60] were the nearest thing around. These implementations of the 23 GoF patterns [48] in both Java and AspectJ were (are) one of the nearest things to examples of good AO style and design, presenting a clear notion of the desirable internal structure for aspects. In practice, they comprised the only available example of code bases available in both a Java and AspectJ forms.

4.2. Approach Taken

Each AspectJ pattern implementation that is different from the Java implementation offers the opportunity to find refactorings capable of transforming the Java solution into the AspectJ solution. Our approach was to characterise the refactorings that would be needed to yield such transformations.

However, no pattern has a single, unique solution, in whatever language. Rather, there a range of solutions from which designers must pick one for their specific problem. No single set of implementations can provide the richness and variety required for our task, and for this reason the Java versions from [60] were regarded as mere a starting point. The prototype refactorings obtained from analysing the aforementioned examples were

subsequently tested and refined with other Java implementations. The refactoring process described in Chapter 7 derives from one of those test sessions.

Fortunately, there are multiples Java implementations of the GoF patterns available in the literature [32][52] and/or posted on the Internet [39][180][178][181]. We collected several of these code bases, and we focused mainly on the ones from [39][32] because they seemed to be the most complete and imaginative. The variety of implementations obtained from different examples further enriched the patterns' potential as providers of insights. Indeed, we believe that our work did not exhaust their potential as provider of insights [122]. For this reason, we mention their continuing exploration in the list of opportunities for further work (see section 9.3).

4.3. Ad Hoc Analysis of the AspectJ Implementations

Though we did not approach the code examples with the intent of undertaking a complete or thorough analysis, we derived various insights from them while carrying out our work on the examples. These include some limitations of the AspectJ implementations or hurdles that can be felt when trying to use them on a concrete situation. We believe these insights may be of interest to researchers and that is why we present them here. Note that this analysis is neither thorough nor comprehensive: it merely describes several scattered insights that we derived during the course of our work. For a more rigorous analysis, we refer to the work by Garcia et al [49] (see also Chapter 8). In section 7.1, we also present a more detailed analysis of the Observer pattern ([48], p. 293), one instance of which provides the subject of the refactoring example presented there.

The analysis presented next focuses on issues and difficulties that may arise in the use of AspectJ constructs, rather than focusing on the patterns. Though we refer to concrete patterns when discussing each of the issues, this is done for illustration purposes. For each of the patterns covered we provide a minimal introduction, as our purpose is not to analyse each pattern in depth – for that, we refer to [48]. Therefore, each description does not necessarily mentions all the roles associated with each of the patterns we cover.

4.3.1 Role Assignment and Role Separation

Of the 23 AspectJ implementations of the patterns, 22 are different from those in plain Java (the identical implementation is that of Façade [48], p.185), offering unique advantages as well as their own trade-offs. Generally, the advantages brought by AspectJ translate into greater modularisation of crosscutting code. Thus, the patterns with the greater degree of crosscutting structure across the various participants see the greater improvement with the use of AspectJ. In our view, the most successful implementation is that of Observer ([48], p.293) and the least successful is that of Façade ([48], p.185), whose AspectJ version is identical to that in Java.

In [60], Hannemann and Kiczales analyse the GoF patterns and their AspectJ implementations according to the roles defined by the pattern and four modularity properties (see Table 4). The pattern roles are classified as:

- *Superimposed* – The object has a distinct, primary role besides the one assigned by the pattern. Superimposed roles can be attached to and detached from existing objects.
- *Defining* – The object exists only to play the pattern role. Such roles do not lend themselves for a separation of roles because the participant plays only that role.

Roles can be defining, superimposed, or something in between. The authors acknowledge that in some cases the classification is not clear-cut.

The four modularity properties are:

- *Locality* – All the code related to the implementation of the pattern is in the abstract and concrete aspects, and none in the participant classes. The participant classes are entirely free of the pattern context, and consequently there is no coupling between the participants. Potential changes to instance of the pattern are confined to a single place.

- *Reusability* – The core pattern code is abstracted and reusable. The implementation of the pattern is generalising the overall pattern behaviour. The abstract aspect can be reused and shared across multiple instances of the pattern. For each pattern instance, we only need to define one concrete subaspect of the abstract aspect.

- *Composition Transparency* – Because a pattern participant's implementation is not coupled to the pattern, if one of the participants takes part in multiple pattern instances, their code does not become more complicated and the pattern instances are not confused. Each instance of the pattern can be reasoned about independently.

- *(Un)pluggability* – Because the participants need not be aware of their roles in any pattern instance, it is possible to switch between using a pattern and not using it in the system.

Table 4 is taken from [60], presenting the results of the above-mentioned analysis. From the table we can see that there are significant differences between the patterns in terms of the nature of the roles. The AspectJ implementations explore the fact that each pattern assigns roles to their participants. AspectJ is able to separate the various roles of a class by placing the code associated to each different role in its own unit of modularity. The technique used for most of the patterns is based on a marker interface that a specific target class is made to implement through a 'declare parents' clause. Patterns that define a role solely for the purposes of the pattern context derive only a limited or no benefit from this capability of AspectJ.

One good example of a defining role can be found in the Command pattern ([48], p.163). The intent of Command is to "encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations." The pattern prescribes that some the objects represent *commands*, usually modelled by an abstract class (C++) or an interface (Java). Concrete commands are instances of concrete classes extending a Command type. The pattern also defines the *invoker* role for objects that place requests to the command objects. In many uses of Command, the command participant exists for the sole purpose of implementing command objects. In Java, it is usual to implement commands through anonymous inner classes, which are by definition non-reusable. Therefore, many instances of Command do not derive much benefit from the stronger compositional capabilities of AspectJ.

On the other hand, one would expect patterns defining superimposed roles to derive much benefit in terms of modularity and reusability. To some extent, we found that to be the case. Observer ([48], p.293) particularly benefits in this respect, because it stands apart in being the only pattern that includes *two* superimposed roles, in addition to not including any defining roles. Consequently, the AspectJ implementation of Observer modularises a whole *collaboration* and that is the reason why it looks particularly impressive (see section 7.1 for a more detailed analysis of Observer). Chain of Responsibility ([48], p.223) bears some

similarities to Observer, but the collaboration between objects is done in terms of a single role. For this reason, the AspectJ implementation of this pattern does not impress so much.

	Modularity Properties				Kinds of Roles	
Pattern name	Locality(**)	Reusability	Composition Transparency	(Un)plug-gability	Defining(*)	Superimposed
Façade	Same implementation for Java and AspectJ				Façade	–
Abstract Factory	no	no	no	no	Factory, Product	–
Bridge	no	no	no	no	Abstraction, Implementor	–
Builder	no	no	no	no	Builder, (Director)	–
Factory Method	no	no	no	no	Product, Creator	–
Interpreter	no	no	n/a	no	Context, Expression	–
Template Method	(yes)	no	no	(yes)	(AbstractClass), (ConcreteClass)	(Abstract Class), (Concrete Class)
Adapter	yes	no	yes	yes	Target, Adapter	Adaptee
State	(yes)	no	n/a	(yes)	State	Context
Decorator	yes	no	yes	yes	Component, Decorator	Concrete-Component
Proxy	(yes)	no	(yes)	(yes)	(Proxy)	(Proxy)
Visitor	(yes)	yes	yes	(yes)	Visitor	Element
Command	(yes)	yes	yes	yes	Command	Commanding, Receiver
Composite	yes	yes	yes	(yes)	(Component)	(Component, Leaf)
Iterator	yes	yes	yes	yes	(Iterator)	Aggregate
Flyweight	yes	yes	yes	yes	FlyweightFactory	Flyweight
Memento	yes	yes	yes	yes	Memento	Originator
Strategy	yes	yes	yes	yes	Strategy	Context
Mediator	yes	yes	yes	yes	–	(Mediator), Colleague
Chain of Responsibility	yes	yes	yes	yes	–	Handler
Prototype	yes	yes	(yes)	yes	–	Prototype
Singleton	yes	yes	n/a	yes	–	Singleton
Observer	yes	yes	yes	yes	–	Subject, Observer

(*) The distinctions between defining and superimposed roles for the different patterns were not always easy to make. In some cases, roles are clearly superimposed (e.g. the Subject role in Observer), or defining (e.g. State in the State pattern). If the distinction was not totally clear, the role names are shown in parentheses in either or both categories.

(**)Locality: "(yes)" means that the pattern is localized in terms of its superimposed roles but the implementation of the remaining defining role is still done using multiple classes (e.g. State classes for the State pattern). In general, (yes) for a desirable property means that some restrictions apply.

Table 4: Analysis of AspectJ implementations of the GoF patterns according to roles and modularity properties

Mediator ([48], p.273) does not seem to benefit from AspectJ's ability to separate roles as much as Observer and Chain of Responsibility, and the reason is that one of its roles is not really superimposed, as is implied in Table 4. The intent of Mediator is to "define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently". The mediator is typically an object acting as the hub of communication for various other objects, named colleagues. In their example, for Mediator Hannemann and Kiczales treat the role of mediator as if it were superimposed, by marking one of the participants with the marker interface that represents the role. However, the object playing the role in the example is an empty shell (it includes only the constructor). The aspect needs a set of the adequate joinpoints to attach the additional state and behaviour and the mediator object serves this purpose. One could say that, in practice, the object is still playing the role, though the associated code is placed separately in the aspect.

We think the AspectJ Mediator (as well as Command and some of the other patterns) is a case of an aspect trying to improve on a code base that is really not amenable to improving (at least on the basis of these designs). It is important to keep in mind that AOP complements a given paradigm rather than replace it. If the base decomposition already manages to encapsulate a role, there isn't much scope for AspectJ to improve things.

4.3.2 A Price for Reusability

Hannemann and Kiczales claim to have localised the implementation of 17 of the patterns, 14 of which also attained transparent composability. They also claim to have produced reusable aspects for 12 of the 17 patterns. In our analysis of these 12 reusable patterns, we detected problems and limitations in some of them. Some of the aspects seem awkward to use, with limitations that may not be noticed until one attempts to use them in concrete cases. Naturally, these examples are to some extent speculative, but since these problems are not mentioned in [60], we mention them in our description. We conclude that in some of the patterns reusability comes with a price, just as it happens with traditional OO technology. In the case of some patterns, we wonder whether the benefits are worth paying. This particularly applies to Composite ([48], p.163) and for this reason we use it to illustrate the problem.

The intent of the Composite pattern ([48], p.163) is to "compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." The main roles defined by the pattern are the *component*, which declares the common interface, the *leaf*, which represents the leaf objects in the composition, and *composite*, defines the behaviour for components that have children, including the storage and management of the children.

The AspectJ implementation of Composite (see Listing 13) attains complete obliviousness from the pattern roles, but the authors aimed to make the aspect reusable as well. The CompositeProtocol aspect implements the composite functionality and defines a small framework comprising:

- Three marker interfaces representing a component, a composite component and a leaf component (Listing 13, lines 2-4). The remaining managing logic is done in terms of these interfaces.
- A private hash table (Listing 13, line 6) responsible for mapping components to their children.

- A protocol based on visitors, through which operations on the elements of the composite structure can be performed. The visitors are objects implementing one of two alternative inner interfaces (Listing 13, lines 28-30 and 37-39), each defining an operation receiving a component as argument and differing in the return type – void for operations (Listing 13, line 29) and Object for functions (Listing 13, line 38).

The requirement that all operations on elements go through the visitor interfaces – with all values passed as a *single* object – places a constraint on the client programmer. For instance, it is hard to implement operations in which an operation on one of the composite's elements use the results from the operation performed on the parent node or on the previous children. The complexity is a result of the efforts to yield a reusable aspect. We experimented writing a case-specific alternative, eschewing the visitors, and the result was significantly simpler. We therefore wonder whether the aspect's reusability sufficiently compensates for the awkwardness in using it. Similar problems plague in various degrees some of the other reusable aspects.

4.3.3 Difficulties in Capturing Context

Our analysis of the code examples also revealed potential difficulties in capturing the context necessary for a given task. That was particularly noticeable in the example for Command. The AspectJ implementation is based on the reusable aspect CommandProtocol that is shown in Listing 14, (without most of the comments so that it fits in one page). The roles defined by Command are modelled by standalone auxiliary interfaces (see Listing 15).

```
01 public abstract aspect CompositeProtocol {
02    public interface Component {}
03    protected interface Composite extends Component {}
04    protected interface Leaf extends Component {}
05
06    private WeakHashMap perComponentChildren = new WeakHashMap();
07
08    private Vector getChildren(Component s) {
09       Vector children = (Vector)perComponentChildren.get(s);
10       if ( children == null ) {
11          children = new Vector();
12          perComponentChildren.put(s, children);
13       }
14       return children;
15    }
16
17    public void addChild(Composite composite, Component component) {
18       getChildren(composite).add(component);
19    }
20    public void removeChild(Composite composite, Component component) {
21       getChildren(composite).remove(component);
22    }
23
24    public Enumeration getAllChildren(Component c) {
25       return getChildren(c).elements();
26    }
27
28    protected interface Visitor {
29       public void doOperation(Component c);
30    }
31    public void recurseOperation(Component c, Visitor v) {
32       for (Enumeration enum = getAllChildren(c); enum.hasMoreElements(); ) {
33          Component child = (Component) enum.nextElement();
34          v.doOperation(child);
35       }
36    }
37    protected interface FunctionVisitor {
38       public Object doFunction(Component c);
39    }
40    public Enumeration recurseFunction(Component c, FunctionVisitor fv) {
41       Vector results = new Vector();
42       for (Enumeration enum = getAllChildren(c); enum.hasMoreElements(); ) {
43          Component child = (Component) enum.nextElement();
44          results.add(fv.doFunction(child));
45       }
46       return results.elements();
47    }
48 }
```

Listing 13: CompositeProtocol – reusable implementation of Composite.

```
01 import java.util.WeakHashMap;
02 import ca.ubc.cs.spl.aspectPatterns.patternLibrary.Command;
03 import ca.ubc.cs.spl.aspectPatterns.patternLibrary.CommandInvoker;
04 import ca.ubc.cs.spl.aspectPatterns.patternLibrary.CommandReceiver;
05
06 public abstract aspect CommandProtocol {
07 // Invoker -> Command mapping //
08    private WeakHashMap mappingInvokerToCommand = new WeakHashMap();
09
10    public Object setCommand(CommandInvoker invoker, Command command) {
11       return mappingInvokerToCommand.put(invoker, command);
12    }
13    public Object removeCommand(CommandInvoker invoker) {
14       return setCommand(invoker, null);
15    }
16    public Command getCommand(CommandInvoker invoker) {
17       return (Command) mappingInvokerToCommand.get(invoker);
18    }
19 // Command -> Receiver mapping //
20    private WeakHashMap mappingCommandToReceiver = new WeakHashMap();
21    public Object setReceiver(Command command, CommandReceiver receiver) {
22       return mappingCommandToReceiver.put(command, receiver);
23    }
24    public CommandReceiver getReceiver(Command command) {
25       return (CommandReceiver) mappingCommandToReceiver.get(command);
26    }
27 // Command Execution via PC & advice //
28    protected abstract pointcut commandTrigger(CommandInvoker invoker);
29    after(CommandInvoker invoker): commandTrigger(invoker) {
30       Command command = getCommand(invoker);
31       if (command != null) {
32          CommandReceiver receiver = getReceiver(command);
33          command.executeCommand(receiver);
34       } else {
35          // Do nothing: This Invoker has no associated command
36       }
37    }
38 // setCommand() via PC & advice //
39    protected pointcut setCommandTrigger(CommandInvoker invoker,
40                                Command command);
41    after (CommandInvoker invoker, Command command):
42          setCommandTrigger(invoker, command) {
43       if (invoker != null) {
44          setCommand(invoker, command);
45       } else {
46          // If the invoker is null, the command cannot be set.
47          // Either ignore this case or throw an exception
48       }
49    }
50 // removeCommand() via PC & advice //
51    protected pointcut removeCommandTrigger(CommandInvoker invoker);
52    after(CommandInvoker invoker): removeCommandTrigger(invoker) {
53       if (invoker != null) {
54          removeCommand(invoker);
55       } else {
56          // If the invoker is null, the command cannot be removed.
57          // Either ignore this case or throw an exception
58       }
59    }
60 // Command default method implementations //
61    public boolean Command.isExecutable() {
62       return true;
63    }
64 }
```

Listing 14: CommandProtocol – reusable implementation of Command.

```
public interface Command {
   public void executeCommand(CommandReceiver receiver);
   public boolean isExecutable();
}
```

```
public interface CommandInvoker {
}
```

```
public interface CommandReceiver {
}
```

Listing 15: Standalone interfaces auxiliary to CommandProtocol.

CommandProtocol provides several variants for associating commands to invokers. The association can be *implicit*, through pointcuts and advice (Listing 14, lines 38-49), or *explicit*, through calls to the aspect method setCommand (Listing 14, lines 10-12). The Main class uses the explicit mode (not necessarily a bad thing). We wondered how the implicit mode could be used and noticed that it presents a technical hurdle related to context capture. CommandProtocol models this functionality through the protected pointcut setCommandTrigger (Listing 14, lines 39-40), which receives both the invoker and the command objects as parameters. We think that in practice, code bases are not likely to expose the context enabling the capture of both the invoker and the command in a single pointcut (the whole point is to enable those separations). The illustrating use case that accompanies the example (see Listing 16) uses the explicit and it is hard to imagine how it could use the implicit mode. Therefore, we suspect the implicit mode will not be feasible in most cases. Theoretically, this hurdle can be circumvented by refactoring the base code, but this is likely to be very invasive, and the resulting changes risk defeating the whole purpose of the refactoring – to make the base code oblivious of the association between invokers and commands.

```
public static void main(String[] args) {
      Button button1 = new Button("Button1");
      Button button2 = new Button("Button2");
      Button button3 = new Button("Button3");

      Command com1 = new ButtonCommand();
      Command com2 = new ButtonCommand2();

      JPanel pane = new JPanel();
      pane.add(button1);
      ButtonCommanding.aspectOf().setCommand(button1, com1);

      pane.add(button2);
      ButtonCommanding.aspectOf().setCommand(button2, com2);
      ButtonCommanding.aspectOf().setReceiver(com2, new Printer());
      pane.add(button3);
      ButtonCommanding.aspectOf().setCommand(button3, com1);

      JFrame frame = new JFrame("Command Pattern Example");

      frame.getContentPane().add(pane);
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {System.exit(0);}
      });
      frame.pack();
      frame.setVisible(true);
   }
}
```

Listing 16: Illustrating use case for the Command pattern.

4.3.4 Problems of Privileged Access

The Memento pattern ([48], p.283) poses the problem of how to provide a given object with privileged access to the internals of another. The intent of Memento is: "without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later. " Memento defines the *originator* as the object whose state must be stored in a snapshot, the *memento* as the object storing a snapshot of the originator's internal state, and the *caretaker* as the object responsible for the memento's safekeeping.

The memento needs privileged access to the originator's internals, something that is tricky to achieve in many languages (Cooper remarks in [32] that this is not directly possible in Smalltalk. Gamma et al. [48] suggest in that C++ implementations use the friend construct). A common solution in Java [32] is to give the default (i.e. package protected) access to the originator's state, so that only classes within the same package have access to it, and place the memento class in the same package as the originator's.

One would expect the AspectJ solution from [60] to rely on a privileged aspect, but it doesn't, probably because privileged aspects are generally regarded as risky and bad style, and/or because its use fails to meet the requirement that encapsulation be preserved. Instead, the AspectJ solution comprises a reusable abstract aspect modelling the role of originator, which must be superimposed on the class of the objects to be safeguarded. The logic for generating the memento is based on a standalone Memento interface that must be used in all cases.

The Memento interface is another instance of awkward and inflexible interfaces due to the attempt to yield reusability. Memento declares a setState operation for passing the state to be safeguarded to the memento, and a getState operation for retrieving it from the memento. In order to be generally applicable, the Memento interface cannot refer to case-specific types. Therefore, the setState operation accepts an argument of type Object and getState returns a value of type Object. Clients of Memento must resort to type-unsafe downcasts. This interface strikes us as too constraining, because originators need to encapsulate their internals within a single object to be passed to setState. This entails creating an *additional* type just for the originator to pass its state to and from mementos. The use case in Main dodges this problem because the originator's state is a simple protected int field, wrapped within an Integer object and upcasted to Object when passed to the memento. Downcasts are used for the memento to return its snapshot.

The Java design presented by Cooper ([32], p.169) implements the memento as a peer class placed within the originator's source file. The fields of the originator have package-protected access. The memento gets the snapshot by receiving the originator as an argument to her constructor. The memento holds a reference to its associated originator and is able to restore the originator's saved state through a restore method. The originator class itself does not contain any code associated with the pattern. In our view, this design compares favourably with the AspectJ design. It achieves the same main advantage as the AspectJ design – the originator is oblivious of its role in the pattern (though at the price of having the memento class in its source file).

4.3.5 Problems with Illegal Weaving

We saw another potential hurdle when we made experiments on implementations of Decorator ([48], p.175). The hurdles relate to the fact that weaving on proprietary libraries usually violates the licenses under which they are provided.

The intent of Decorator is to "attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." Decorators are a way to emulate mixins [22] in languages not directly providing the mechanism. The pattern defines the decorator participant and one maintaining a reference to the decorated object – the *component* – and exporting an interface that conforms to that of the component. The *component* participant is the object to be decorated with additional functionality and which defines the common interface. The AspectJ implementation is based on advice – Hannemann and Kiczales [60] remark on the inherent limitations of this approach, namely the dynamic reordering of decorators. Advices are less flexible due to their static (i.e. compile time) nature.

When we tried to use the same approach to the example of Decorator presented in [32], we discovered we couldn't, because the component is the JButton type from Java's javax.swing standard API. This prevented us from using pointcut designators such as execution on the component operations because this would entail weaving on the Java's jar file exporting the JButton type. Such weaving violates the license under which Java is made publicly available. In situations such as these, programmers must resort to the call pointcut designator, which targets the binaries making the calls, rather than the code that is actually executed. However, in the example from [32] the JButton object is called within a JFrame object from the same API. It would be also illegal to weave at either "side" of the component.

4.4. Results Derived from the Code Examples

The analysis undertaken on the code examples described in this Chapter yielded most of the material documented in the next two chapters, which includes all the refactorings with the exception of those mentioned in Chapter 3, which are:

- *Partition Constructor Signature* (137)
- *Move Field from Class to Inter-type* (104)
- *Move Method from Class to Inter-type* (106)
- *Extract Fragment into Advice* (94)
- *Extract Feature into Aspect* (90).

All the aforementioned refactorings were updated during our work on the present case study, with the exception of *Partition Constructor Signature* (137). The material for the validating example presented in Chapter 7 also originates from the case study described in this Chapter.

4.5. Summary

In this section, we summarise the main insights we derived from the work on the code examples:

- Each of the 23 GoF design patterns assigns roles to the various participants. The primary advantage of AspectJ over OO languages is in the ability to separate the secondary roles from the primary functionality of the participants objects. However, in many cases the pattern role is the only role played by a participant (i.e. the role is defining). In such cases, the improvements brought by AspectJ are limited or non-existent. The advantage of AspectJ is particularly noticeable when a pattern assigns more than one superimposed role. In such cases, AspectJ is able to

modularise the entire collaboration based on the superimposed roles. Observer is the only example among the 23 GoF patterns of a modularised collaboration.

- AOP does not seem to be different from previous techniques in that there is a price to pay for reusability. Though AspectJ manages to modularise certain concerns to an extent that is beyond the each of OO, this does not mean that the resulting aspect is reusable. Attaining reusability in some cases resulted in added complexity as well as interfaces that are inflexible and awkward to use. In previous sections we describe a few examples. Just as with OO, there is a trade-off between reusability, and flexibility and user-friendliness.
- Aspects need an adequate base of joinpoints to leverage their quantification capabilities. In a few cases, methods and objects were left in the code base whose purpose seemed solely to provide a convenient joinpoint to the aspect. These situations tended to occur when the nature of the pattern role (superimposed vs. defining) was not clear-cut.
- There are limits to the abilities of AspectJ to capture the necessary context. A case is described in which the parameter list of an abstract pointcut required the capture of more different values than those that seem to be readily available from existing joinpoints.
- The capabilities of AspectJ to weave onto proprietary libraries raise issues and difficulties whose nature is legal rather than technical. A case was described in which AspectJ could not be used because using it would entail weaving into the jar files that ship with the Java install. This would be in violation of the license under which Sun Microsystems distributes the Java files.

Chapter 5. Aspect-Oriented Code Smells

This Chapter presents the code smells that were derived from work on the two case studies presented in the two previous chapters. Bad smells in the source code are indicators of undesirable situations that should be removed or corrected, and the purpose of refactorings is to do just that. For this reason, the descriptions of the smells include suggestions of the refactorings that can remove them. These refactorings are presented in Chapter 6 and therefore the present Chapter includes many references to material found in the following Chapter. This way, we establish the links between the smells and the refactorings.

The smells are of two kinds: (1) to detect the presence of crosscutting concerns in OO code, and (2) to detect a scope for improvement in aspect-oriented code. Most of the smells presented here are of the former kind. The last smell – *Aspect Laziness* – is of the latter kind and can only occur in aspects.

Smells aiming to detect the presence of crosscutting concerns in OO code can be divided into traditional OO smells seen from a new, aspect-specific perspective, and novel smells, specifically designed to assist in the detection of crosscutting concerns. It should be noted that when we refer to "OO code" in the context of detecting crosscutting concerns, we do not imply that the code must necessarily be free of aspect code. In this context, we simply mean that the "base code" (i.e. parts of a system comprising classes and methods) may still include crosscutting concerns that are amenable to being extracted to aspects.

This Chapter is structured as follows. Section 5.1 surveys traditional OO smells in the light of AOP. In section 5.2 the *Double Personality* is proposed. The *Abstract Classes* smell is proposed in 5.3, and the *Aspect Laziness* smell is proposed in 5.4.

5.1. OO Smells in the Light of AOP

From our analysis of the code smells presented in [47], [158] and [76], we concluded that some can be used by AO programmers as symptoms of the presence of crosscutting concerns. This particularly applies to the following two OO smells:

- *Divergent Change* ([47], p.79) –The ideal situation when we want to change something in a system is to be able to pinpoint a single clear point where the change can be made. The Divergent change smell is felt when we can't do this. According to Beck and Fowler, divergent change occurs when one class is commonly changed in different ways for different reasons. As an example, the authors mention the situation of a programmer looking at a class and saying "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument". In this example, two objects are likely to be better than one. However, note that it may not be worth the programmer's effort (unless it is a small one and does not add much complexity) to prepare the code to avoid the changes, if she does not expect to add new databases[19]. On the other hand, if the programmer regularly adds new financial instruments, she will clearly benefit from removing the smell.
- *Shotgun Surgery* ([47], p.80) – As Beck and Fowler point out, shotgun surgery is similar to divergent change but is the opposite. This smell is felt when we have to

[19] Beck and Fowler define the *Speculative Generality* ([47], p.83) smell to represent situations in which the code is needlessly prepared (and made more complex) for changes that are not certain to be required.

> make many little changes in many different classes every time one needs to make a kind of change to the system. When the changes are scattered everywhere, they are hard to find and it is easy to miss an important change.

Beck and Fowler sum up the two smells by saying that "shotgun surgery is one change that alters many classes and divergent change is one class that suffers many kinds of changes". These two smells are clearly symptoms of code scattering code tangling respectively. Wake [158] mentions configuration information, logging and persistence as possible causes to the *Shotgun Surgery* smell, all of which can be counted among the favourite examples for the use of AOP. Clearly, they can be used to signal the presence of crosscutting concerns. However, we think it is useful to be attentive to these symptoms in methods as well, i.e. "one change that alters many methods and one method that suffers many kinds of changes" can signal the presence of crosscutting concerns, even if restricted to the internals of one class. Such a concern cuts across the members of a single class (or a single class hierarchy). Laddad [91] cites class-level contract enforcement, lazy initialisation of resources and class-specific exception handling as examples of such concerns.

Kerievsky [76] proposes a variant of *Shotgun Surgery* that he calls *Solution Sprawl* ([76], p.43) – "you become aware of this smell when adding or updating a system feature causes you to make changes to many different pieces of code". Kerievsky stresses the similarity of the two smells but notes that they are sensed differently – "we become aware of *Solution Sprawl* by observing it, while we detect *Shogun Surgery* by doing it". Both variants are equally promising as indicators of crosscutting concerns.

The *Extract Feature into Aspect* (90) refactoring was defined as a general framework for the extraction of the fragments and members related to the kind of concern that these smells help to detect.

5.2. The Double Personality Code Smell

The *Double Personality* smell can be found in classes that play multiple roles. Ideally, each class should play a single role, meaning that it contains only one, coherent, set of responsibilities. This often is not possible in OO frameworks and applications.

Examples of *Double Personality* can be found in the OO implementations of design patterns [48] that include what Hannemann and Kiczales call *superimposed roles* – roles assigned by the pattern to classes that have functionality and responsibility outside the pattern [60].

As an example of *Double Personality*, consider the Chain of Responsibility ([48], p.223) pattern. Its purpose is to allow a chain of objects to attempt to handle a request, without any of them knowing about the capabilities of the other objects, which can be of different types. Whenever an object receives a request that it cannot handle, it passes to the next object in the chain. How the system behaves when the end of the chain is reached depends on the specific case. The pattern provides a loose coupling between the handler classes, the only common link being the logic they all must have to pass the request between them. This common logic is modelled by the Handler role, which is superimposed on most of the participant classes.

Listing 17 shows one of the classes from the implementation of Chain of Responsibility, taken from [32]. The Handler role is modelled by the interface Chain[20], which all handler participants must implement. Listing 17 shows one such participant, where the code related to the role of handler is shaded.

[20] "Chain" is arguably an inadequate name for the interface. A name such as "Handler" would be more intentional.

```
public interface Chain {
   public abstract void addChain(Chain c);
   public abstract void sendToChain(String mesg);
   public Chain getChain();
}
```

```
public class ColorImage extends JPanel implements Chain {
   private Chain _nextChain;
   public ColorImage() {
      super();
      setBorder(new LineBorder(Color.black));
   }
   public void addChain(Chain c) {
      _nextChain = c;
   }
   public void sendToChain(String mesg) {
      Color c = getColor(mesg);
      if (c != null) {
         setBackground(c);
         repaint();
      } else {
         if (_nextChain != null)
            _nextChain.sendToChain(mesg);
      }
   }
   private Color getColor(String mesg) {
      String lmesg = mesg.toLowerCase();
      Color c = null;
      if (lmesg.equals("red"))
         c = Color.red;
      if (lmesg.equals("blue"))
         c = Color.blue;
      if (lmesg.equals("green"))
         c = Color.green;
      return c;
   }
   public Chain getChain() {
      return _nextChain;
   }
}
```

Listing 17: Example of the *Double Personality* smell

More examples of *Double Personality* can be found in other patterns. In [60], Hannemann and Kiczales present an analysis of the GoF patterns covering the superimposition of roles. The various classes from the refactoring example that we present in Chapter 7 – based on the Observer pattern ([48], p.293) – also betray strong doses of *Double Personality* (both the Subject and Observer roles are superimposed on various participants).

The example from Listing 17 also suggests that the implementation of interfaces is a useful symptom that can help to detect *Double Personality* in Java sources. Interfaces are a popular way to model roles in Java – e.g. use of interfaces to model roles comprise the motivation for *Extract Interface* ([47], p.341). When a class implements an interface modelling a role that does not relate to the class' primary concern, the class smells of *Double Personality*. Indeed, this insight forms the very basis of the work by Tonella and Ceccato [153] (see Chapter 8).

Double Personality can lead to the detection of two different situations: a secondary role superimposed on a *single* class, and a role cutting across *multiple* classes. If *Double Personality* is detected in one class, we suggest that developers analyse the code base to confirm that it applies to just that class. Again, looking to the interfaces may help: if multiple classes implement the interface, this means the secondary concern is crosscutting.

If only one class is affected, or if the code of the secondary role is restricted to the implementation of the interface, the solution is to extract the secondary role to a mixin [22]. There are several ways to do this. Laddad's *Extract Interface Implementation* [91] suggests a refactoring process that results on the secondary functionality being extracted to a mixin. The mixin is implemented through an inner aspect enclosed within the interface which models the superimposed role. However, this implementation seems to assume that the implements clause should remain in original implementing class. Consequently, the coupling between the class and the mixin remains explicit. In some cases, the programmer may strive for total obliviousness of the secondary role, in which case we suggest using *Replace Implements with Declare Parents* (108) after applying *Extract Interface Implementation*.

Extract Interface Implementation is suitable for creating mixins, but not for dealing with more complex cases. As an alternative to the implementation proposed by Laddad, we propose *Split Abstract Class into Aspect and Interface* (109). This refactoring also extracts the secondary role but it leads to a different implementation of the extracted mixin that is based on a standalone aspect. *Split Abstract Class into Aspect and Interface* (109) has the advantage that the resulting aspect can develop and grow without trouble, if it turns out that the concern is more complex than what can be enclosed within a mixin. The implementation that results from using *Split Abstract Class into Aspect and Interface* (109) is similar to some of the implementations used by Hannemann and Kiczales to implement the GoF patterns (e.g. Chain of Responsibility, Mediator and Observer).

If the analysis of the code performed after detecting *Double Personality* reveals that the code related to the crosscutting concern is more complex than a simple implementation of an interface, we suggest using *Extract Feature into Aspect* (90) to move all the related elements to an aspect.

5.3. Abstract Classes as a Code Smell

The AspectJ composition mechanisms enabling the emulation of mixins also enable the separation of definitions (i.e. implementation code) from declarations in abstract classes, so that these can be turned into interfaces. Hannemann and Kiczales take this approach in implementing five of the GoF design patterns in AspectJ [60]. This separation has the advantage that classes become free to inherit from some other class, and we still have the option of introducing a default implementation to the interfaces. This suggests that abstract classes should be considered a code smell, at least in the cases when they prevent us from inheriting from some other desirable class. Two of the refactorings documented in Chapter 6 remove that smell by moving implementation code to an aspect and turning abstract classes into interfaces. The *Split Abstract Class into Aspect and Interface* (109) refactoring can be used to extract the concrete members of an abstract class into an aspect, and we can turn the resulting pure abstract class into an interface using *Change Abstract Class to Interface* (88).

5.4. The *Aspect Laziness* Code Smell

The *Aspect Laziness* smell applies to aspects that do not carry the full weight of their responsibilities and instead pass the burden to classes, in the form of inter-type declarations. We detect this smell in aspects that resort to the mechanism of inter-type declarations to add state and behaviour to a class when something more dynamic and/of flexible would be desirable. The problem with inter-type declarations is the fact that they are a static mechanism. Inter-type declarations apply to all instances of the target class, throughout their entire life cycles. For this reason, inter-type declarations are inadequate for

solving problems that require a degree of flexibility. Inter-type declarations should be avoided (or refactored) in any of the following cases:

- The additional state and/or behaviour are needed by only a subset of the instances of the target classes.
- The additional state and/or behaviour are needed only during certain specific phases in the execution of the program.
- Instances of the target classes may simultaneously require multiple instances of the additional state and behaviour.

In any of the above cases, the mechanism of inter-type declarations is not dynamic or flexible enough. In such cases, it is preferable to provide the aspect with the logic that manages the mapping between the target objects and the additional state and behaviour. Hannemann and Kiczales showed in the code they presented in [60] how to meet these flexibility requirements in an elegant way. According to their designs, the aspect owns a data structure mapping the specific instances to the extra state. In many cases, the mapping functionality can be implemented using an instance of one of Java's collections (usually a hash table), plus a few aspect methods to manage it. The *Replace Inter-type Field with Aspect Map* (118) and *Replace Inter-type Method with Aspect Method* (125) refactorings were specifically designed to replacE the existing inter-type declarations with a mapping logic that provides the same functionality in a more flexible way.

Examples of *Aspect Laziness* can be found in Chapter 7. These are the two aspects resulting from the extraction of crosscutting concerns. Subsection 7.7.2 shows how the smell can be removed. Listing 17 shows one of the aspects resulting from the extraction process (described in section 7.7.1). The example comprises one implementation of the Observer pattern ([48], p.293) involving three participant classes – Flower, Bee and Hummingbird (see section 7.2 for a description of the original example). All the members found in the aspect from Listing 17 originated from members of the three participant classes. The extractions were performed as prescribed by the various refactorings for extracting elements related to a crosscutting concern (these are described in section 6.3).

Aspect Laziness can be clearly detected thanks to the inter-type declarations of fields (Listing 18, lines 34-36) and methods (Listing 18, lines 38-46). Note that the functionality provided by the aspect does not meet any of the flexibility requirements mentioned above. The internal structure of the extracted aspect still reflects the original OO design, which decentralises the associated state and behaviour throughout the participant objects. This contrasts with the very different design shown in Listing 19. There, the same functionality is concentrated within the aspect. In this case, it is implemented by means of a hash map (Listing 19, line 14) and the various methods that manage it (Listing 19, lines 24-44). The aspect shown in Listing 19 corresponds to the design presented by Hannemann and Kiczales in [60]. Only a few more refactorings are needed to extract it from the aspect shown in Listing 19 (these are described in subsection 7.7.3).

The aspect shown in Listing 18 also betrays the *Duplicated Code* ([47], p.76) smell. Both the openObsrv field (Listing 18, lines 35 and 36) and the openObserver method (Listing 18, lines 41-43 and 44-46) are declared twice. The reason is that they are related to the Observer role in the pattern, of which there are two in this example. The same would happen to the oNotify field (Listing 18, line 34) and the opening method (Listing 18, lines 38-40) if there were more than one Subject participant. It is also important to note that the extraction process exposed duplication that could not be removed using traditional OO techniques – the duplication would be unavoidable and scattered. With AO techniques, this

duplication is simply exposed as a code smell – *Duplicated Code* ([47], p.76) – which can be removed with further refactorings.

```
01 import java.util.Observable;
02 import java.util.Observer;
03
04 public aspect ObservingOpen {
05    private interface BreakfastTaker {
06       public void breakfastTime();
07    }
08    declare parents: (Bee || Hummingbird) implements BreakfastTaker;
09
10    static class OpenNotifier extends Observable {
11       private Flower _enclosing;
12       private boolean alreadyOpen = false;
11       public OpenNotifier(Flower flower) {
12          _enclosing = flower;
13       }
14       public void notifyObservers() {
15          if(_enclosing.isOpen() && !this.alreadyOpen) {
16             this.setChanged();
17             super.notifyObservers();
18             this.alreadyOpen = true;
19          }
20       }
21       public void close() {
22          this.alreadyOpen = false;
23       }
24    }
25    static class OpenObserver implements Observer {
26       private BreakfastTaker _enclosing;
27       public OpenObserver(BreakfastTaker enclosing) {
28          _enclosing = enclosing;
29       }
30       public void update(Observable ob, Object a) {
31          _enclosing.breakfastTime();
32       }
33    }
34    private OpenNotifier Flower.oNotify = new OpenNotifier(this);
35    private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
36    private OpenObserver Bee.openObsrv = new OpenObserver(this);
37
38    public Observable Flower.opening() {
39       return oNotify;
40    }
41    public Observer Bee.openObserver() {
42       return openObsrv;
43    }
44    public java.util.Observer Hummingbird.openObserver() {
45       return openObsrv;
46    }
47    pointcut flowerOpen(Flower flower):
48       execution(void open()) && this(flower);
49    after(Flower flower) returning : flowerOpen(flower) {
50       flower.oNotify.notifyObservers();
51    }
52    pointcut flowerClose(Flower flower):
53       execution(void close()) && this(flower);
54    after(Flower flower): flowerClose(flower) {
55       flower.oNotify.close();
56    }
57 }
```

Listing 18: Code Example illustrating the *Aspect Laziness* smell.

```
01 import java.util.WeakHashMap;
02 import java.util.List;
03 import java.util.ArrayList;
04 import java.util.ListIterator;
05
06 public aspect ObservingClose {
07    private interface Subject {}
08    private interface Observer {}
09
10    public abstract boolean Subject.isOpen();
11    public abstract void Observer.bedtimeSleep();
12    private boolean Subject.alreadyClosed = false;
13
14    private WeakHashMap subject2ObserversMap = new WeakHashMap();
15
16    private List getObservers(Subject subject) {
17       List observers = (List)subject2ObserversMap.get(subject);
18       if(observers == null) {
19          observers = new ArrayList();
20          subject2ObserversMap.put(subject, observers);
21       }
22       return observers;
23    }
24    public void addObserver(Subject subject, Observer observer) {
25       List observers = getObservers(subject);
26       if(!observers.contains(observer))
27          observers.add(observer);
28       subject2ObserversMap.put(subject, observers);
29    }
30    public void removeObserver(Subject subject, Observer observer) {
31       getObservers(subject).remove(observer);
32    }
33    public void clearObservers(Subject subject) {
34       getObservers(subject).clear();
35    }
36    private void notifyObservers(Subject subject) {
37       if(!subject.isOpen() && !subject.alreadyClosed) {
38          subject.alreadyClosed = true;
39          List observers = getObservers(subject);
40          for(ListIterator it = observers.listIterator(); it.hasNext();) {
41             ((Observer)it.next()).bedtimeSleep();
42          }
43       }
44    }
45    pointcut flowerOpen(Subject subject):
46       execution(void open()) && this(subject);
47    after(Subject subject) returning : flowerOpen(subject) {
48       subject.alreadyClosed = false;
49    }
50    pointcut flowerClose(Subject subject):
51       execution(void close()) && this(subject);
52    after(Subject subject): flowerClose(subject) {
53       notifyObservers(subject);
54    }
55
56    declare parents: Flower implements Subject;
57    declare parents: (Bee || Hummingbird) implements Observer;
58 }
```

Listing 19: Code Example after removing the *Aspect Laziness* smell.

Chapter 6. Aspect-Oriented Refactorings

This Chapter describes the refactorings derived from the case studies presented in chapters 3 and 4. Some of the refactorings are mentioned in Chapter 5, in the context of code smells. Though the main source of insights comprised implementations of design patterns, all the refactorings presented here – with one possible exception, the one presented in section 6.6 – are generally applicable. The exception, or special case, is a refactoring that deals with a situation that can arise in legacy code with published interfaces. The collection of refactorings is structured in three groups, plus the special case.

The refactorings documented in this Chapter do not attempt to cover all possible situations that can potentially arise in source code. For instance, they do not cover uses of reflection. Likewise, they do not deal with what we call the *fragile base code problem* [118][89] i.e. the fact that almost any refactoring can potentially break existing pointcuts. For instance, *Move Method from Class to Inter-type* (106) can break pointcuts using the within pointcut designator. We believe human programmers will be able to deal thoroughly with these issues only when provided with a new generation of tools, specifically designed to account for the presence of aspects. However, we also believe it is possible to keep these problems under control provided the adequate practices are followed, including programming AspectJ's constructs with a prudent and appropriate style (see section 6.2), such as that proposed by Laddad [91].

This Chapter is structured as follows. Section 6.1 presents the format used to describe the refactorings. Section 6.2 presents some guidelines that can be applied in most refactoring processes. Section 6.3 describes the group of refactorings that deal with the extraction of crosscutting concerns from plain Java code bases to aspects. Section 6.4 describes the refactorings that relate to the improvement of the internal structure of aspects, particularly aspects extracted through the refactorings from the previous group. Section 6.5 describes the refactorings that deal with factoring out common code found in multiple aspects.

6.1. Format of the Refactorings

We aimed to present the refactorings in a format that programmers could recognise, to enhance their immediate applicability. For this reason we present them in such a way that they become a natural extension of the one used by Fowler et al. (Kerievsky took the same approach in [76]). The refactorings were also given names in the same style, and the presentation format is likewise similar. This format is demanding to the developer of the refactorings, because it requires much detail in both the mechanics and in code examples. However, it greatly benefits the programmer, who is thus provided with a clear idea of when the refactoring should be applied, and how to apply it.

The format includes the following items:

- Name of the refactoring.
- Brief mention of a typical situation.
- Brief description of the recommended action.
- Motivation, sometimes complemented with a Pre-conditions section.
- Mechanics.

- Code example(s) (except in the simplest refactorings).

In some refactorings, we complement the motivation section with a preconditions section to clarify the scope of applicability. The mechanics sections emphasize the safe way of performing a refactoring, just like the mechanics described by Fowler et al [47]. Throughout the descriptions, we cross-reference the refactorings, mentioning the page number. Likewise, whenever a refactoring from Fowler's book [47] is mentioned, we mention both the reference and the page number. The purpose of the code examples is to illustrate the use of each refactoring in the proper context. They are not meant to be self-contained and, just as in the examples presented in Fowler's book, the resulting code at the end is not necessarily problem-free. We also follow the example of Fowler et al in highlighting the changed code in bold and deleted code ~~crossed like this~~, in the situations where it helps to make the changes easier to spot.

Following the example of Fowler's book, the descriptions of the refactorings avoid mentioning code smells[21].

6.2. General Guidelines

Some guidelines are applicable to most refactoring processes, independently of the transformation being carried out. Techniques such as the following are essential, considering that presently there is no adequate tool support for AO refactorings.

The first guideline is to ensure the programs are adequately unit tested [182][47] before applying manual code transformations. Good test coverage remains as essential to the refactoring process as before, if no more so.

A second guideline is to ensure the base code exposes all the desirable joinpoints. An effective way to do this is to ensure the code adheres to the style advocated in Fowler's book [47], namely placing each relevant concept in its on class, and the use of small methods with meaningful names. Fewer and bigger classes with long methods make it more likely that necessary context will take the form of local method variables rather than object fields, thus preventing the necessary context from being available for capture at the appropriate moment. Long methods make it more likely that AspectJ will not be able to insert the additional behaviour in the desirable place within the method. Besides making the system easier to understand, using the appropriate style also enriches the system with the potentially desirable joinpoints (e.g. more beginnings and ends of methods, more method parameters, more return values, more object fields, etc).

A third guideline relates to pointcuts, which should be made in a style stressing intent [84] (typically expressions using wildcards) rather than a specific case. This way, pointcuts can express a general policy and may be robust enough to not being affected by minor modifications in the target code, such as the removal or addition of a new class or method. Another good practice is to place the aspects close to the code they affect whenever possible, to increase the likelihood that all team members are aware of the aspects potentially affected by refactorings. This often entails placing the aspect in the same package, or even within the same source file as the target class (as inner or peer aspects). Though the mechanics sections warn of a few potential problems, we expect programmers to rely on their knowledge of the code base to check potential problems.

[21] This is cause for some duplication between the description of the smells and the of the motivation texts of some refactorings. Our aim is to make the description of each refactoring as self-contained as possible.

In [91] Laddad prescribes several guidelines to ensure the refactorings are applied in a safe way. These involve the creation of a first version of a pointcut through the case-by-case enumeration of each interesting joinpoint, followed by its subsequent refactoring to replace it with a semantically more meaningful pointcut [84]. To assist in this refactoring, Laddad provides a recipe based on AspectJ's 'declare error' mechanism to verify whether two different pointcuts p1 and p2 capture exactly the same set of joinpoints:

```
declare error:  (!p1() && p2()) || (p1() && !p2()):
   "Mismatch in join points captured";
```

Prior to extracting crosscutting behaviour from large systems, Laddad also proposes the use of advice deleting the behaviour about to be extracted (through around advice without calls to proceed), providing another opportunity to monitor the effects of the aspect.

Though the refactorings do not mention specific tools, the mechanics sections assume the user of the refactorings is using a modern IDE with support for AspectJ (such as eclipse/AJDT), providing services such as structure views, various kinds of searches and the existing support for plain Java refactorings. These services, particularly structure views, are essential for any refactoring process targeting large and complex systems. These services can be complemented by the use of 'declare warning' clauses, which the mechanics sections prescribe at the appropriate points. From our experience, 'declare warning' clauses can be very effective in speeding up a manual refactoring process involving multiple scattered points, particularly if the IDE supports aspects (e.g. eclipse [179] with AJDT [169]) and is capable of opening files and going to the relevant point in the code with a simple mouse click on the line stating the warning. Considering that the code related to the concerns targeted by AOP is by definition scattered throughout multiple units of modularity, all these services play an important role.

The original OO refactorings can be used in AspectJ code as well. Throughout our work, we did not detect any refactoring from [47] targeting a specific construct that could not be applied to that construct within aspects. For instance, we prescribe the use of *Extract Method* ([47], p.110) to code within aspects in the mechanics of *Extend Marker Interface with Signature* (113).

6.3. Refactorings for Feature Extraction

The refactorings presented in this section deal with moving the various implementation elements from their original places into aspects. They comprise the starting point of any refactoring process dealing with plain Java code bases. Our experience suggests these will be the most frequently used refactorings. This group is headed by *Extract Feature into Aspect* (90), a composite refactoring that covers the general feature extraction algorithm. The remaining refactorings from the first group refine its various steps.

The "open class" mechanism of inter-type declarations makes it particularly easy to move elements to aspects. From the point of view of client code, there is no difference between a public method declared in its own class or introduced by an aspect. There is no need to scan client code in search for potential problems the move may have caused: we know from the start it didn't cause any. The same ease applies, to varying degrees, to other elements.

The extraction of methods and fields to aspects, as well as the extraction of snippets of code to advice, are probably among the most "obvious" refactorings. As it would be expected, several proposals for such refactorings have been presented [73][58]. We also present our own proposals: *Move Method from Class to Inter-type* (106) to move methods to

aspects, *Move Field from Class to Inter-type* (104) to move fields to aspects, and *Extract Fragment into Advice* (94) to move a code snippet to an advice in the aspect.

However, our work taught us that the elements of implementation can go beyond just methods, fields and sequences of statements. The limitations in OOP sometimes lead programmers to use inner classes to cope with code duplicated in multiple classes. OOP does not enable programmers to modularise the code as much as they would like, but they still can use inner classes to tidy up the internals of each of the classes involved, ant attain a better separation between the secondary code from the one related to the primary concern. With AOP, we can go further, using *Extract Inner Class to Standalone* (99) to turn the inner class into a standalone class, and next using *Inline Class within Aspect* (102) to place it inside the aspect.

A frequent technique to organise the code and make intentions clear is to use interfaces to represent roles played by classes. With AOP, we can completely modularise the secondary role to which the interfaces are associated. Sometimes the interfaces continue to be useful as maker interfaces within an aspect. Interfaces can be inlined within the aspect using *Inline Interface within Aspect* (103), as well as the connections with their implementing classes, using *Replace Implements with Declare Parents* (108). Sometimes the interface is not an interface at all, but an abstract class, which includes some concrete elements. These definitions can be separated into an aspect using *Split Abstract Class into Aspect and Interface* (109), after which we can use *Change Abstract Class to Interface* (88) to turn the resulting pure abstract class into an interface.

Interestingly, two of the refactorings documented in this section relate to plain Java constructs, but their motivation arises only in the context of aspects – *Change Abstract Class to Interface* (88) and *Extract Inner Class to Standalone* (99).

6.3.1 Change Abstract Class to Interface

Typical situation

An abstract class prevents their subclasses from inheriting from another class.

Recommended action

Turn the abstract class into an interface and change its relationship with the subclasses from inheritance to implementation.

Motivation

This refactoring will be typically used when someone is using *Split Abstract Class into Aspect and Interface* (109) in an abstract class. It comprises the final step of turning it into an interface; to be performed after all concrete members were moved to an aspect. By then the class should be a pure abstract class (i.e. an abstract class without any concrete methods and non-static fields). Theoretically, it could be argued that a pure abstract class should always be turned in an interface, for this makes it plain that we are not in the presence of any concrete elements. This way all implementing classes will be free to extend some other class if there is a need to. However, we do not propose such a strong statement and defer that issue to future work.

If the abstract class inherits from another class, this refactoring may change type relationships. Depending on the specific relationships, this might break the source code or not, and you must check for potential problems. If you see no problems, start applying this refactoring to the highest class of the inheritance chain and then proceed downwards.

This refactoring will be possible only if all classes in the inheritance chain are owned by the programmer. In addition, if there are concrete classes higher up this refactoring cannot be applied. At the very least, some restructurings in the inheritance chain must be performed first.

Keep in mind that all signatures declared in the resulting interface must be public, as well as all the corresponding methods of implementing classes.

Mechanics

- Change the keywords 'abstract class' to 'interface'.
- If the abstract class implements some interfaces, change the 'implements' keyword to 'extends'.
- You can optionally remove the keyword 'abstract' from the method declarations.
- Update all classes inheriting from the abstract class. For each subclass, change the keyword from 'extends' to 'implements'. If the subclass implements other interfaces, just remove the 'extends' keyword and move the name of the former abstract class to the list of implemented interfaces.

Example

See also the example for *Split Abstract Class into Aspect and Interface* (109).

```
public abstract class Builder {
   public abstract void processType(String type);
   public abstract void processAttribute(String type);
   public abstract void processValue(String type);
   public abstract String getResult();
}
```

```
public class StructureBuilder extends Builder {
   //...
}
```

```
public class TextBuilder extends Builder {
   //...
}
```

```
public interface Builder {
   public abstract void processType(String type);
   public abstract void processAttribute(String type);
   public abstract void processValue(String type);
   public abstract String getResult();
}
```

```
public class StructureBuilder implements Builder {
   //...
}
```

```
public class TextBuilder implements Builder {
   //...
}
```

6.3.2 Extract Feature into Aspect

Typical situation

Code related to a feature is scattered across several methods and classes, tangled with unrelated code.

Recommended action

Extract all the implementation elements related to the feature to an aspect.

Motivation

The most usual elements of implementation are methods and fields, which are the ones we usually move across modules when performing refactorings. However, some programming languages provide support for other kinds of members: for instance, Java allows for inner classes and interfaces. The catalogue proposed in [47] does not account for these constructs, probably because it wouldn't make much sense, due to its nature. Inner classes and interfaces comprise implementation elements that are tightly coupled to the rest of the implementation: for instance, inner classes can access the private members of their enclosing classes. They are not meant to be part of the interface of modules, and instead are used to provide a better internal structure of a class. Consequently, those elements often make sense only within their current enclosing classes.

The motivation for using inner classes and interfaces is often a direct consequence of the limitations in Java's composition capabilities. Often, they are used as an attempt to compensate for the crosscutting effect caused by an additional concern. Inner classes help to better separate some parts of a class from "other parts", which programmers would ideally place in separate modules[22], if these could be created. However, when we have AOP's superior composition capabilities we have the option of doing exactly that. Likewise, interfaces are used to compensate for the scattering and duplication effects caused by secondary concerns[23] whose code must be duplicated in every affected module, causing tangling in those modules.

With AspectJ, we can design structures that would not be possible with plain Java, and therefore it makes sense to extract, move and inline certain kinds of members that otherwise would not be considered, including inner classes and interfaces. After modularisation such members may no longer be needed, in which case they will be removed or transformed when dealing with the internals of the new modules (probably using transformations prescribed by *Tidy Up Internal Aspect Structure* (128). That should be done only after extracting all the related elements, when we can take advantage of the new modularity. This issue falls outside the scope of this refactoring but it is important to include a brief mention here, to stress that not everything is done as soon as we move a crosscutting concern to an aspect.

Mechanics

- Create an empty aspect in the appropriate package.
- If you find inner classes related to the extracted concern use *Extract Inner Class to Standalone* (99) to each of them. You can later use *Inline Class within Aspect* (102).

---

[22] Eckel's flower example for the Observer pattern ([48], p.293)[1], used in the validating example from Chapter 7, is a good example of this: its extensive use of inner classes is meant to compensate for the lack of mixin inheritance [22].

[23] Independent authors reached this same conclusion [153] regarding the use of interfaces.

- Move the concern's various fields to the aspect with *Move Field from Class to Inter-type* (104). Since fields are usually private, you may have to temporarily declare the aspect as privileged in order to keep the code compilable and testable.
- Move initialisation code placed within the constructors using *Extract Fragment into Advice* (94). If some of that code uses some of the constructor's parameters, first restructure that part of the code. Consider using *Extract Method* ([47], p.110) to replace the parameter in the constructor and related code with a separate method. In cases in which the constructor is part of a published interface that cannot be changed, consider using *Partition Constructor Signature* (137).
- Move the concern's various methods to the aspect with *Move Method from Class to Inter-type* (106).
- When only part of the method relates to the concern there are two options: (1) use *Extract Method* ([47], p.110) and then *Move Method from Class to Inter-type* (106), or (2) use *Extract Fragment into Advice* (94). When the fragment uses an argument from the enclosing method, it may be simpler to use the latter.
- Apply *Inline Class within Aspect* (102) to any former inner classes that are used only within the aspect. Likewise, apply *Inline Interface within Aspect* (103) to any interfaces that are no longer used outside the aspect.
- Change to private the access modes of all aspect members that are now used only within the aspect.
- Remove the qualifier privileged from the aspect as soon as it no longer accesses non-public members in the primary code.

After extracting all the elements from the primary code, consider using *Tidy up Internal Aspect Structure* (128) to improve the internal structure of the resulting aspect.

Example: Extracting Two Concerns from a Tangled Stack

Here we provide a small complete example (initially presented in [118]) that illustrates some of the issues of extracting features from an existing class and helps to show how some refactorings fit in the larger picture. No client program is presented here, but care was taken to ensure that the refactorings are transparent to client code.

The initial example, shown in Listing 20, comprises a FIFO structure plus two crosscutting concerns: (1) support to a simple window view of stack's state and (2) precondition checking. This is a case where the responsibility for checking preconditions lies in the client, and that is why the exception used here is unchecked. We do not present the definition of the runtime exception, as it is quite trivial.

We start by extracting the window view concern from the base code, by applying *Extract Feature into Aspect*. We first create an empty aspect *WindowView* and then move all members related to this concern. These include two fields, *_label* and *_text*, so we start with these, by applying *Move Field from Class to Inter-type* (104) to each in turn.

Both field transfers require similar sequences of steps: (1) copy the declaration of the field to the aspect, (2) add 'TangledStack.' before the field's name, (3) delete (or comment out) the field's original declaration, (4) when moving the first field include the declaration 'import javax.swing.*;' in the import section of the aspect, and (5) change the field's access to public. Compile and test after moving each field.

The initialisation code for both fields should be transferred next. The constructor receives an argument (the JFrame object) related to the extracted concern, so *Partition Constructor*

Signature (137) is used. This results in two versions of the constructor. One version is argument less and is placed in the host class, dealing only with the primary concern. The other version, receiving the JFrame object that relates to the extracted concern, is placed in the aspect, and therefore made (un)pluggable. As this constructor should not include any code unrelated to its concern, it includes a call to super rather than duplicate the other initialisation code.

```
import javax.swing.*;

public class TangledStack {
   private int _top = -1;
   private Object[] _elements;
   private final int S_SIZE = 10;
   private JLabel _label = new JLabel("Stack ");
   private JTextField _text = new JTextField(20);

   public TangledStack(JFrame frame) {
      _elements = new Object[S_SIZE];
      frame.getContentPane().add(_label);
      _text.setText("[]");
      frame.getContentPane().add(_text);
   }
   public String toString() {
      StringBuffer result = new StringBuffer("[");
      for(int i=0;i<=_top;i++) {
         result.append(_elements[i].toString());
         if(i!=_top)
            result.append(", ");
      }
      result.append("]");
      return result.toString();
   }
   private void display() {
      _text.setText(toString());
   }
   public void push(Object element) {
      if(isFull())
         throw new PreConditionException("push when stack full.");
      _elements[++_top] = element;
      display();
   }
   public void pop() {
      if(isEmpty())
         throw new PreConditionException("pop when stack empty.");
      _top--;
      display();
   }
   public Object top() {
      if(isEmpty())
         throw new PreConditionException("top when stack empty.");
      return _elements[_top];
   }
   public boolean isFull() {
      return (_top == S_SIZE-1);
   }
   public boolean isEmpty() {
      return (_top<0);
   }
}
```

Listing 20: Initial illustrating example for *Extract Feature into Aspect*.

We use *Move Method from Class to Inter-type* (106) to move the method display, and we use *Extract Fragment into Advice* (94) to move the calls to display in the push and pop methods to a piece of advice. The declaration 'import javax.swing.*;' can be removed from the host

class at this point. Finally, we can change to private the access qualifiers of the two moved fields and method.

Extraction of the precondition checking concern is similarly performed according to *Extract Feature into Aspect*, though this case is simpler, comprising three executions of *Extract Fragment into Advice* (94) for the tests in push, pop and top, respectively. After both aspects are created as described, the host class is as shown in Listing 21 and the two aspects should look as shown in Listing 22 and Listing 23.

```
public class TangledStack {
   private int _top = -1;
   private Object[] _elements;
   private final int S_SIZE = 3;
   public TangledStack() {
      _elements = new Object[S_SIZE];
   }
   public String toString() {
      StringBuffer result = new StringBuffer("[");
      for(int i=0;i<=_top;i++) {
         result.append(_elements[i].toString());
         if(i!=_top)
            result.append(", ");
      }
      result.append("]");
      return result.toString();
   }
   public void push(Object element) {
      _elements[++_top] = element;
   }
   public void pop() {
      _top--;
   }
   public Object top() {
      return _elements[_top];
   }
   public boolean isFull() {
      return (_top == S_SIZE-1);
   }
   public boolean isEmpty() {
      return (_top<0);
   }
}
```

Listing 21: Clean code after using *Extract Feature into Aspect.*

```
import javax.swing.*;

public aspect WindowView {
   private JLabel TangledStack._label =
      new JLabel("Stack ");
   private JTextField TangledStack._text = new JTextField(20);
   public TangledStack.new(JFrame frame) {
      this();
      frame.getContentPane().add(_label);
      _text.setText("[]");
      frame.getContentPane().add(_text);
   }
   private void TangledStack.display() {
      _text.setText(toString());
   }
   pointcut stateChange(TangledStack stack):
      (execution(public void stack.TangledStack.push(Object))
       ||
       execution(public void stack.TangledStack.pop()))
      && this(stack);
   after(TangledStack _this) returning :
      stateChange(_this) {
      _this.display();
   }
}
```

Listing 22: Window view concern after extration into an aspect.

```
public aspect PreConditionChecking {
   pointcut checkPush(TangledStack stack):
      execution(public void
         TangledStack.push(Object))
      && this(stack);
   before(TangledStack _this): checkPush(_this) {
      if(_this.isFull())
         throw new PreConditionException("push when stack full");
   }
   pointcut checkPop(TangledStack stack):
      execution(public void TangledStack.pop())
      && this(stack);
   before(TangledStack _this): checkPop(_this) {
      if(_this.isEmpty())
         throw new PreConditionException("pop when stack empty");
   }
   pointcut checkTop(TangledStack stack):
      execution(public Object TangledStack.top())
      && this(stack);
   before(TangledStack _this): checkTop(_this) {
      if(_this.isEmpty())
         throw new PreConditionException("top when stack empty");
   }
}
```

Listing 23: Precondition checking concern after extraction to an aspect.

6.3.3 Extract Fragment into Advice

Typical situation

Part of a method is related to a concern whose code is being moved to an aspect.

Recommended action

Create a pointcut capturing the required joinpoint and context and move the code fragment to an appropriate advice based on the pointcut.

Motivation

This refactoring should be used when we want to move to an aspect a piece of functionality that does not comprise a complete method. Sometimes it is a simple method call. In some cases it is convenient to turn the part that should be moved into its own method, using *Extract Method* ([47], p.110]), and next use *Move Method from Class to Inter-type* (106) on it. However, even in such occasions there will be need to create an advice that calls the method.

Before copying the code fragment, a careful analysis of the method's (or, sometimes, the constructor's) body should be made, in order to find a suitable pointcut to capture the exact set of intended joinpoints. If the primary code does not offer a suitable joinpoint, one or more refactorings must be performed until the code is ripe for this refactoring.

Sometimes the advice will need to capture local variables (either primitives or object references). These are a problem, because AspectJ cannot capture the values of local variables. Note that these situations may be a sign that the method is more complicated than it should be. Consider whether it would make sense to split it in various parts, using *Extract Method* ([47], p.110]) for each part in turn. Such a split may provide the joinpoints you need. For instance, the arguments or return value of one of extracted methods may expose the context that was previously available only in a local variable.

In the more awkward cases when even the above options are of no avail, use *Replace Method with Method Object* ([47], p.135). This is the refactoring recommended by Fowler et al. to ease the way for *Extract Method* ([47], p.110), but it may be even more appropriate to the present case, for it is almost certain to provide you with the missing leverage for capture of context. This solution is preferable to crudely turning the local variable into a field: this way, the lifetime of the fields of the method object is restricted to the execution of the method, rather than to the lifetime of the original object. However, keep in mind that it may not be possible to keep the fields of the method object private: consider using *Encapsulate Field* ([47], p.206). In addition, *Replace Method with Method Object* ([47], p.135) cannot be used in constructors and recursive methods.

If the fragment to be extracted uses an internal type, consider first using *Extract Inner Class to Standalone* (99) on that type before applying this refactoring.

Mechanics

- Create a named pointcut that captures the intended set of joinpoints. If the intended pointcut already exists (from previous uses of *Extract Fragment into Advice*), change it to include the joinpoint needed for the new fragment.
- Make sure that the pointcut captures the context required by the code fragment. In particular, check if the extracted fragment mentions 'this' or 'super', or includes self-calls. In such cases, a reference to the executing object must be captured. Choose a suitable name for the variable holding the captured object. In some cases, the choice may be straightforward. In others, use a general yet meaningful name such as '_this' or 'self'.
  Example (from the illustrating example used for *Extract Feature into Aspect*(90)):

```
pointcut stateChange(TangledStack stack):
    execution(public void TangledStack.push(Object))
    && this(stack);
after(TangledStack _this) returning : stateChange(_this) {
    _this.display();
}
```

- Check whether all types used in the pointcut are known to the aspect. In some cases new import declarations may need to be added (this applies even when the type is mentioned only in pointcuts).
- Create the suitable advice for the pointcut, with an empty body (in case it is not already under construction).
- Move the code to extract from the source method into the advice's body.
- Replace references to the self-variable ‘this’ by the corresponding variable obtained from the context capture.
- Scan the extracted code for references to any variables that are local in scope to the source method, including parameters and local variables. Declarations of any temporary variables used only within the extracted code can be placed inside the advice's body. In some cases, you may need to make some adjustments to set up the advice’s context.

When the advice is meant to replace a large number of scattered fragments you should choose the simpler of the two options: (1) to deal with the whole set at a single go, or (2) to deal with one fragment at a time. Sometimes the pointcut is complicated to specify when covering only a subset of the intended joinpoints. If that is the case, you may consider writing the full, intended pointcut right at the start. The drawback then is that you'll have to factor all the scattered fragments to the common advice at a single go before you can compile and test again. You should avoid this whenever the scattered fragments are not identical or very similar (e.g. calls to the same method). In some cases, it may be worthwhile to refactor the various fragments so that they become more alike (e.g. giving the same names to locals and parameters) and therefore easier to reason with. Ideally, the fragments could be turned into calls to a common method, using *Extract Method* ([47], p.110).

Example: Simple Extraction of a Method Call

```
public class TangledStack {
   //...
   public void push(Object element) {
      _elements[++_top] = element;
      display();
   }
```

```
public class TangledStack {
   //...
   public void push(Object element) {
      _elements[++_top] = element;
      display();
   }
```

```
pointcut stateChange(TangledStack stack):
   execution(public void TangledStack.push(Object))
   && this(stack);

after(TangledStack _this) returning :
   stateChange(_this) {
   _this.display();
}
```

Example: Complex example requiring prior refactoring to make the code aspect-friendly

The following case arose during the refactoring experiment described in Chapter 3 [118], which involved the complete extraction of a concern in order to make it pluggable. At a given time, we wanted to use *Extract Fragment into Advice* on a method called RepeatUntilProcedure.compile, but the extracted code needed various local values from the previous part of the method for its computation. The values were stored neither as fields of some object nor as the return value (the method was overlong). The structure of the code was as sketched below:

```
public class RepeatUntilProcedure ... {
   //...
   public Graph compile() {
      // 17 LoCs
      if (...)
         // one LoC
      else {
         //13 LoCs related to the primary concern
         //66 LoCs related to the data link concern
      }
      //a few more LoCs related to the data link concern
   }
   return result;
}
```

```
public aspect DataLinksAspect {
   //...
}
```

Using *Extract Method* ([47], 110) on the chunk of interest, we make it easier to reason with.

```
public class RepeatUntilProcedure ... {
   //...
   private boolean compileDataLinks(<various arguments>) {
      // 66 LoCs
      return result;
   }
   public Graph compile() {
      // 17 LoCs
      if (...) {
         //1 LoC to prepare for the call to compileDataLinks()
         ... = compileDataLinks(...);
         //A few more LoCs
      }
      return result;
   }
```

Next, we used *Move Method from Class to Inter-type* (106) to the compileDataLinks method:

```
public class RepeatUntilProcedure ... {
   //...
   public Graph compile() {
      // 17 LoCs
      if (...) {
         //1 LoC to prepare for the call to compileDataLinks()
         ... = compileDataLinks(...);
         //A few more LoCs
      }
      return result;
   }
```

```
public aspect DataLinksAspect {
   //...
   //private
   public
   boolean RepeatUntilProcedure.compileDataLinks(...) {
      // 66 LoCs
      return result;
   }
}
```

We then applied *Replace Method with Method Object* ([47], p.135) to compileDataLinks. The class of the method object was implemented with the Compile inner class within RepeatUntilProcedure:

```
public class RepeatUntilProcedure ... {
   //...
   public class Compile {
      private final RepeatUntilProcedure _enclosing;
      //several fields stemming from the local variables of compileDataLinks()
      public Compile(RepeatUntilProcedure repeatUntilProc) {
         _enclosing = repeatUntilProc;
      }
      public Graph compute() {
         //The same 17 LoCs of compile(), with the following differences:
         // ° the code uses the fields instead of the former locals
         // ° some references to this were replaced by _enclosing
         if (...) {
            //1 LoC to prepare for the call to compileDataLinks()
            ... = compileDataLinks(...);
            //A few more LoCs related to the data link concern
         }
         return _result;
      }
   }
   //...
```

We were finally ready to apply *Extract Fragment into Advice*, leading to the following layout:

```
public class RepeatUntilProcedure ... {
   //...
   public class Compile {
      //...
      public Graph compute() {
         //17 LoCs
         if (...) {
            //1 LoC to prepare for the call to compileDataLinks()
            ... = compileDataLinks(...);
            //A few more LoCs related to the data link concern
         }
         return _result;
      }
   }
   //...
```

```
public aspect DataLinksAspect {
   //...
   private boolean compileDataLinks(<various arguments>) {
      // 66 LoCs
      return result;
   }
   pointcut repeatUntilProcComputeCompile(
         RepeatUntilProcedure _this,
         RepeatUntilProcedure.Compile compile):
      call(public Graph RepeatUntilProcedure.Compile.compute())
      && target(compile)
      && this(_this);

   /** Add processing of data-links at the end of the RepeatUntilProcedure
compile. */
   after(RepeatUntilProcedure _this, RepeatUntilProcedure.Compile compile)
         returning: repeatUntilProcComputeCompile(_this, compile) {
      if (...) {
         //1 LoC to prepare for the call to compileDataLinks()
         ... = _this.compileDataLinks(compile, hasFeedBack);
         //A few more LoCs related to the data link concern
      }
   }
}
```

The refactoring is complete. Notice that the compileDataLinks method could be made private again, this time to the aspect. In circumstances such as these, the code inside the aspect is likely to need some tidying up.

6.3.4 Extract Inner Class to Standalone

Typical situation

An inner class relates to a concern being extracted into an aspect.

Recommended action

Eliminate dependencies from the enclosing class and turn the inner class into a standalone class.

Motivation

Even if Java programmers are unable to decouple a class from certain accessory functionalities, they can at least place those functionalities in a well-localised way, separate from the rest of the class's code. Inner classes can be used to structure the internals of a class and localise functionalities not related to the primary functionality of the class, thereby separating it from the rest of the class's implementation. This is facilitated by the capability of inner classes to refer to members of the enclosing object, including the private ones. To a limited extent, inner classes can be used as if they were *subclasses* of its enclosing class, with the advantage that they can still inherit from another, unrelated, class.

However, when we have AOP's superior composition capabilities this may no longer be the best available option. The kinds of secondary functionality found within inner classes are best modularised within aspects, and that in turn provides the motivation for extracting them, something that would probably not make sense with plain Java.

Turning inner classes into standalone classes is a preparatory step for *Inline Class within Aspect* (102). Note that the limitation of Java's composition capabilities makes it likely that the functionality placed in the extracted classes is duplicated in multiple classes. In such cases, applying this refactoring will expose that duplication, particularly when several inner

classes have the same name. Moving scattered elements within an aspect is very effective to expose various kinds of duplication, including of this kind.

Mechanics

- Look for any code within the inner class relating to behaviour that should be kept within the enclosing class. Use *Extract Method* ([47], p.110) on those parts.
- Create in the inner class a private field of the type of the enclosing class.
  In some cases, the definition of the inner class is repeated within several enclosing classes, which may not be related by a common type relationship. In such cases first create an interface exposing the common interface used within the inner class and make the enclosing classes implement that interface. Make the inner class's private field of that interface type. This way, a common inner class can be extracted from all the enclosing classes. If you intend to inline the inner classes within an aspect, the interface should be inlined as well – using *Inline Interface within Aspect* (103) – but only after you inline all the implementing classes.
- Create a public constructor for the inner class and include a parameter for the enclosing object providing the initial value of the new field. Update any code related with the creation of instances of the inner class in the enclosing class. If the inner class is not private, also check possible uses outside the enclosing class.
- Compile and test.
- Look for all direct references to fields belonging to the enclosing object. Use *Self Encapsulate Field* ([47], p.171) on all such fields.
- Look for any calls to private methods made within the inner class. Relax the access rules of those methods.
  If you are reluctant to expose those members, you still have the option of doing it only on a temporary basis. As soon as the class is inlined within an aspect, reclassify them as private. However, in this case the aspect will have to be privileged.
  You can make sure that no members are forgotten by placing 'this.' before each field reference and method call (Java does not allow the use of 'this' within inner classes to refer to members of the enclosing class). This is easy to do for inner classes, since they are typically small (otherwise that could be a case for applying *Extract Class* [47], p.149).
- Compile and test.
- Create a standalone class with the same name as the inner class. Copy the source text of the inner class to the standalone class and add 'public' before the class's name. Check for imports that should go along with the class. Check for imports that the host class no longer needs, and delete them.
- Compile and test.
- Delete the inner class. Compile and test again.

Example

The following example uses a simplified version of the Flower class from Eckel's flower example for the Observer pattern ([48], p. 293), also used in Chapter 7. The purpose is to turn the inner class OpenNotifier into a standalone class.

```
public class Flower {
   private boolean _isOpen;
   private OpenNotifier _oNotify = new OpenNotifier();

   public Flower() {
      _isOpen = false;
   }
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      _isOpen = true;
      _oNotify.notifyObservers();
   }
   public Observable opening() {
      return _oNotify;
   }
   private class OpenNotifier extends Observable {
      private boolean _alreadyOpen = false;
      public void notifyObservers() {
         if(_isOpen && !_alreadyOpen) {
            setChanged();
            super.notifyObservers();
            _alreadyOpen = true;
         }
      }
      public void close() {
         _alreadyOpen = false;
      }
   }
```

First, create the field _enclosing and a constructor receiving an argument to initialise that field. The creation of an OpenNotifier object needs to be updated (the compiler can be very useful in this kind of situations):

```
public class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean _alreadyOpen = false;
   public OpenNotifier(Flower enclosing) {
      this._enclosing = enclosing;
   }
   public void notifyObservers() { //...
```

```
public class Flower {
   private boolean _isOpen;
   private OpenNotifier _oNotify = new OpenNotifier(this);
```

Next, replace direct references to fields from the enclosing class. There is one in this example, _isOpen:

```
   boolean isOpen() {
      return this._isOpen;
   }

private class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean _alreadyOpen = false;
   public OpenNotifier(Flower enclosing) {
      _enclosing = enclosing;
   }
   public void notifyObservers() {
      if(_enclosing.isOpen() && !_alreadyOpen) {
```

Next, ensure that all references to member within the inner class have either 'this.', 'super.' or '_enclosing.' (you could also use 'this._enclosing.', of course):

```
private class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean _alreadyOpen = false;
   public OpenNotifier(Flower enclosing) {
      this._enclosing = enclosing;
   }
   public void notifyObservers() {
      if(this._enclosing.isOpen() && !this._alreadyOpen) {
         this.setChanged();
         super.notifyObservers();
         this._alreadyOpen = true;
      }
   }
   public void close() {
      this._alreadyOpen = false;
   }
}
```

Finally, create a standalone class with the same name, copy the contents and delete the inner class.

```
import java.util.Observable;

public class OpenNotifier extends Observable {
   //...
```

6.3.5 Inline Class within Aspect

Typical situation

A small standalone class is used only by code within an aspect.

Recommended action

Move the class to within the aspect.

Motivation

This situation can occur when one or several small helper classes relate to a concern that is being modularised into an aspect. It also occurs during the process of moving an inner class from class to an aspect, after applying *Extract Inner Class to Standalone* (99).

Inner classes are typically small and this refactoring is recommended only for small classes. As a rule of thumb, a class with more than a screenful of lines of code clearly does not qualify as "small". In such cases, consider leaving it as standalone or further refactoring it.

Mechanics

- Create a copy of the class' source code, inside the aspect. Replace public with static just before the class' name.
- Add in the aspect's import section any imports that may be needed by the class.
- Compile and test.
- Delete the standalone class. Compile and test.

Example

We take the example of the class OpenNotifier, from Eckel's flower example for the Observer pattern ([48], p. 293), also used in Chapter 7. We start at the point after OpenNotifier was made a standalone class.

```
import java.util.Observable;

public class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean _alreadyOpen = false;
   public OpenNotifier(Flower enclosing) {
      this._enclosing = enclosing;
   }
   public void notifyObservers() {
      if(this._enclosing.isOpen() && !_alreadyOpen) {
         this.setChanged();
         super.notifyObservers();
         this._alreadyOpen = true;
      }
   }
   public void close() {
      this._alreadyOpen = false;
   }
}
```

The class can be copied almost "as is" to within the body of the aspect, only the first keyword being changed:

```
static class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean _alreadyOpen = false;
   public OpenNotifier(Flower enclosing) {
      this._enclosing = enclosing;
   }
   public void notifyObservers() {
      if(this._enclosing.isOpen() && !_alreadyOpen) {
         this.setChanged();
         super.notifyObservers();
         this._alreadyOpen = true;
      }
   }
   public void close() {
      this._alreadyOpen = false;
   }
}
```

In this case, the code using OpenNotifier needs to import java.util.Observable. If the aspect does not include that import already, it must be added as well. After compiling and testing, this refactoring is complete.

6.3.6 Inline Interface within Aspect

Typical situation

One or several interfaces are used only by an aspect.

Recommended action

Move the interfaces to inside the aspect.

Motivation

This situation usually arises when a feature is being extracted from the existing code base. Often, the feature assigns various roles to various participants using an interface to model each role. As related code is being moved to an aspect, at some point only the aspect knows about the interfaces. When that happens, there is no compelling reason for keeping them standalone.

Sometimes the system relies on a set of interdependent interfaces with each interface declaring methods that refer to other interfaces in their parameter lists. Ideally, we would inline one interface at a time, leaving the code in a compilable state and passing all the tests before inlining the next interface. However, this may not be possible, due to interdependencies. In such cases it is easier to move the whole set of interfaces at one go.

Mechanics

- Create within the aspect a private copy of the standalone interface.
- Delete the standalone interface[24].
- Compile and test.

Example

In the following example, a Mediator[25] aspect is being created to encapsulate the interaction between various Button objects. Interfaces GUIColleague and GUIMediator are used only within the Mediator aspect. Moving only one of the interfaces leads to compiler errors, so they are moved at one go.

```
public interface GUIMediator {
   public void colleagueChanged(GUIColleague colleague);
}
```

```
public interface GUIColleague {
   public void setMediator(GUIMediator mediator);
}
```

```
public aspect Mediator {
   declare parents: Button implements GUIColleague;
   declare parents: Label implements GUIMediator;
   //...
```

```
public aspect Mediator {
   private interface GUIMediator {
      public void colleagueChanged(GUIColleague colleague);
   }
   private interface GUIColleague {
      public void setMediator(GUIMediator mediator);
   }
```

6.3.7 Move Field from Class to Inter-type

Typical situation

A field relates to a concern other than the primary concern of its enclosing class.

Recommended action

Move the field from the class to the aspect as an inter-type declaration.

Motivation

If the field is of an internal type, use *Extract Inner Class to Standalone* (99) on that type before applying this refactoring. The mechanics of this refactoring include some follow-up steps to deal with code that uses the moved field.

---

[24] To play safe, make sure that binary *.class files are not left in the directory associated with the package.

[25] The code used in this example was taken from the Mediator example of [60].

Mechanics

- Copy the declaration of the field from the class to the aspect, including the assignment of an initial value, if one exists. Add the host class's name and '.' before the name of the field in the inter-type declaration.
- Check whether a new *import* statement should be written in the aspect's *import* section, to bring the field's type into its scope. If the aspect is placed in a separate package, also check for the declaration of the host class.
- If the field's access is private, relax it. If the aspect is placed within the same package as the host class, it can be package-protected, otherwise it must be *public*. You will be able change it back to private as soon as all code using the field is placed in the aspect.
- Delete the field's declaration in the host class. Check for any *import* statements that are no longer necessary in the original host class.
- Check for pointcuts using the within pointcut designator that refer to the moved field.
- Compile and test.
- Create a 'declare warning' to signal all occurrences of the moved member in the code, as shown in the following example:

```
public aspect ThisAspect {
    //...
    declare warning:
       (get(JTextField TargetClass._text) || set(JTextField
TargetClass._text))
       && !within(ThisAspect):
       "Field _text is accessed outside aspect.";
    //...
}
```

- For each fragment of code using the field, decide whether the whole method, or just a fragment, should be moved to the aspect: (a) if the whole method must be moved, use *Move Method from Class to Inter-type* (106). (b) If just a fragment should be moved, use *Extract Fragment into Advice* (94). (c) If a parameter of a method or constructor relates to the moved field use *Extract Method* ([47], p.110) to isolate the part that uses the parameter, move it to the aspect with *Move Method from Class to Inter-type* (106), move the call to a more suitable point, and then use *Remove Parameter* ([47], p.277). Use *Replace Inter-type Method with Aspect Method* (125) to deal with inter-type methods that use the inter-type fields. Compile and test after each refactoring.
- As soon as the last access to the field outside the intended scope is removed, restrict again the field's access. This is usually means private, but if for some reason you need to keep in the class some code related to the field, consider using *Encapsulate Field* ([47], p.206). If the field was originally protected and subclasses of the target class need to access it, use a variant of *Introduce Aspect Protection* (116).

Example

```
//...
import javax.swing.*;

public class TangledStack {
   private int _top = -1;
   private Object[] elements;
   private final int S_SIZE = 10;
   private JLabel _label = new JLabel("Stack ");
   private JTextField _text = new JTextField(20);
   //...
}
```

```
public class TangledStack {
   private int _top = -1;
   private Object[] elements;
   private final int S_SIZE = 10;
   private JLabel _label = new JLabel("Stack ");
   private JTextField _text = new JTextField(20);
   //...
}
```

```
import javax.swing.*;

public aspect WindowView {
   public JLabel TangledStack._label = new JLabel("Stack");
   public JTextField TangledStack._text = new JTextField(20);
   //...
}
```

When all the code relative to the fields is placed in the aspect, change their access qualifiers back to private, compile, and test again:

```
import javax.swing.*;

public aspect WindowView {
   private JLabel TangledStack._label = new JLabel("Stack");
   private JTextField TangledStack._text = new JTextField(20);
   //...
}
```

6.3.8 Move Method from Class to Inter-type

Typical situation

A method belongs to a concern other than the primary concern of its owner class.

Recommended action

Move the method into the aspect encapsulating the secondary concern, as an inter-type declaration.

Motivation

If the method contains some logic that should remain in the class, first apply *Extract Method* ([47], p.110).

If the method uses an internal type for its return value or any of its arguments, use *Extract Inner Class to Standalone* (99) on that type before applying this refactoring.

The most straightforward case of moving a method to an aspect is when the method is public, there is only one implementation of its signature throughout the inheritance chain,

and it uses only (1) its parameters, (2) public members, (3) local variables, (4) members already moved from the class to the aspect which are (perhaps temporarily) qualified as public. If these conditions are not all met, some of the following cases should be considered.

a) Check if the method uses any non-public member that may not be visible in the aspect. Consider whether these should also belong to the aspect's concern. If you think they belong to the aspect, consider whether they would be best moved together or one at a time. In some cases, several members may be tightly coupled and would be easier to move together (or it would be a case for using *Extract Class* [47], p.149). In case you want to move them one at a time, start with the fields, applying *Move Field from Class to Inter-type* (104), next move initialisation code in the constructors[26] with *Extract Fragment into Advice* (94), and then move the methods with this refactoring.

b) If the method uses non-public members that you think should remain in the class, check if there are public accessor methods you can use, or if it is worth to create them now (even if just temporarily), or if you can relax the accesses. See also if it is a case of moving the aspect to the same package. If you are reluctant to use any of these options, you'll have to declare the aspect as privileged.

c) A situation where a method needs to access non-public members in both the host class and the aspect may be an indication that the method is addressing more than one concern. If it is the case, the best solution is probably to leave the method in the class, keeping the code relative to the main functionality, and moving the part related to the target concern, using *Extract Fragment into Advice* (94).

d) If the moved method is non-public, check if the methods that call the moved method also belong to the same concern, the same way as in a). Naturally, this is feasible only when just a few methods and fields are involved.

e) Search for any implementations of the same signature in sub- and super-classes. In case you find some, these alternative implementations should belong to the aspect as well, in order to make the related functionality pluggable.

f) A full inheritance hierarchy in the primary code may be a sign that the concern already aligns well with the dominant decomposition. Check if that is the case, and whether the hierarchy should be left in the primary code. If yes, extract only the relevant subset of the code from each of the points of the hierarchy, using *Extract Fragment into Advice* (94).

Mechanics

- Copy the method's definition to the aspect. Add the class name and '**.**' before the name of the method.
- If the access is non-public, it may need to be (temporarily) relaxed. If the aspect is placed within the same package as the host class, package-protected is enough. Otherwise, you must make it public.
- Check whether new import statements should be included in the aspect's import section. Check if some existing import statement is no longer necessary in the host class, and delete them.
- Check for pointcuts referring to the moved method using the within pointcut designator.

---

[26] The refactoring *Partition Constructor Signature* was designed for a special case that can arise here, if the constructor is part of a published interface that the developers are unable or unwilling to modify.

- Delete the method's definition in the class.
- Compile and test.
- The following steps apply if you want to make the method private to the aspect. Add a 'declare warning' to signal all calls to the method, as shown in the following example:

```
public aspect ThisAspect {
    //...
    declare warning:
       (call(<type> <host class>.someMethod(<arguments>))
       && !within(ThisAspect):
       "method <host class>.someMethod() called outside ThisAspect.";
    //...
}
```

- As soon as all the code that uses the method is in the aspect, change it to private. In case of protected access, leave a 'declare error' as prescribed in *Introduce Aspect Protection* (116).

Example: moving a private method.

```
public class TangledStack {
   private void display() {
      _text.setText(toString());
   }
   //...
}
```

```
public class TangledStack {
   private void display() {
      _text.setText(toString());
   }
   //...
}
```

```
public aspect WindowView {
   //...
   public //private
   void TangledStack.display() {
      _text.setText(toString());
   }
}
```

6.3.9 Replace Implements with Declare Parents

Typical situation

Classes implement an interface related to a secondary concern. Implementation of the interface is used only when the related concern is present in the system.

Recommended action

Replace the 'implements' in the class with a 'declare parents' in the aspect.

Motivation

Interfaces are the standard way in Java to represent the various roles played by a class. AspectJ makes it possible to encapsulate many of these roles within aspects, which often resort to marker interfaces to represent them. It is likely that in the process of moving the role to the aspect all references to the interface will be moved as well. When that happens, the interface should also be inlined, using *Inline Interface within Aspect* (103), and turned into a

marker interface. The references to the interface include the 'implements' declarations in the various implementing classes, and these must be inlined as well.

Mechanics

- Create the suitable 'declare parents' in the aspect.
- In the implementing class, delete the 'implements' clause related to the interface.
- Compile and test.

Example: Moving the First Implements Clause

```
class SomeImplementingClass implements TargetInterface, ... {
   //...
```

```
class SomeImplementingClass implements TargetInterface, ... {
   //...
```

```
public aspect Implementation {
   declare parents: SomeImplementingClass implements TargetInterface;
   //...
}
```

Example: Moving the First Implements Clause

When the aspect already has at least one 'declare parents' related to that interface, switch to the following notation:

```
class AnotherClass implements TargetInterface, ... {
   //...
```

```
class AnotherClass implements TargetInterface, ... {
   //...
```

```
public aspect Implementation {
   declare parents: (SomeImplementingClass || AnotherClass)
      implements TargetInterface;
   //...
}
```

6.3.10 Split Abstract Class into Aspect and Interface

Typical situation

Classes are prevented from using inheritance because they already inherit from an abstract class defining some concrete members.

Recommended action

Move all concrete members from the abstract class to an aspect. You can then turn the abstract class into an interface.

Motivation

All classes implementing the interface will inherit the members introduced by the aspect and still be able to inherit from another class.

Mechanics

- Create an aspect that will enclose the concrete members of the abstract class.

- Use *Move Field from Class to Inter-type* (104) to move each field in turn from the abstract class to the aspect.
- When the class includes constructors, you may find two different situations, depending on whether the initialisations use or not values passed to the constructor. If no constructor arguments are required, use *Extract Fragment into Aspect* to move initialisation code from the class's constructor(s) to the aspect. If the initialisation requires arguments, the best option is probably to create setter methods for the fields that need external values and refactor the code so that it uses the setter methods instead of constructors. The setters (and associated getters) can use lazy initialisation of the related fields.
  Introducing a constructor from the aspect is not an option, because the class is going to be turned into an interface, loosing the ability to have constructors.
- For each concrete method, use *Move Method from Class to Inter-type* to move it from the abstract class to the aspect, keeping in the class an abstract declaration the method's signature.
- Use *Change Abstract Class to Interface* (88) to turn the abstract class into an interface and update the subclasses.

Example

The following example is the Java implementation of the *Factory Method* pattern ([48], pp.107-116) by Hannemann and Kiczales [60]. It is worth it to reproduce some comments found in the code regarding this implementation:

"In this example, the factory method createComponent creates a JComponent (a button and a label, respectively). The anOperation method showFrame uses the factory method to show a little GUI. In one case, the created frame contains a button, in the other a simple label.
Since the anOperation method requires an implementation, Creator has to be an abstract class (as opposed to an interface). Consequently, all ConcreteCreators have to be subclasses of that class and cannot belong to a different inheritance hierarchy."

```
import java.awt.Point;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public abstract class Creator {
   private static Point lastFrameLocation = new Point(0, 0);
   public abstract JComponent createComponent();
   public abstract String getTitle();
   public final void showFrame() {
      JFrame frame = new JFrame(getTitle());
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) { System.exit(0); }
      });
      JPanel panel = new JPanel();
      panel.add(createComponent());
      frame.getContentPane().add(panel);
      frame.pack();
      frame.setLocation(lastFrameLocation);
      lastFrameLocation.translate(75, 75);
      frame.setVisible(true);
   }
}
```

First, we create the blank aspect CreatorImplementation. Next, we use *Move Field from Class to Inter-type* (104) to move the field to aspect.

```
import java.awt.Point;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public abstract class Creator {
   private static Point lastFrameLocation = new Point(0, 0);
   public abstract JComponent createComponent();
   public abstract String getTitle();
   public final void showFrame() {
      //...
   }
}
```

```
import java.awt.Point;

public aspect CreatorImplementation {
   private static Point Creator.lastFrameLocation = new Point(0, 0);
}
```

We then use *Move Field from Class to Inter-type* (104) to move the lastFrameLocation field, which was temporarily made package-protected. Next, we use *Move Method from Class to Inter-type* (106) to move the showFrame method, but in this case leaving an abstract declaration of the signature in the abstract class. The lastFrameLocation field can be private again.

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public abstract class Creator {
   public abstract JComponent createComponent();
   public abstract String getTitle();
   public abstract void showFrame();
}
```

```
import java.awt.Point;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public aspect CreatorImplementation {
   private static Point Creator.lastFrameLocation = new Point(0, 0);
   public final void Creator.showFrame() {
      //...
   }
}
```

Next, we use *Change Abstract Class to Interface* (88) to turn Creator into an interface. This entails removing the static classification of the lastFrameLocation field. In many cases, this is not behaviour preserving, and it would require some case-specific workaround. As this isn't much of a problem in this particular case, we leave it as shown:

```
import javax.swing.JComponent;

public interface Creator {
   public abstract JComponent createComponent();
   public abstract String getTitle();
   public abstract void showFrame();
}
```

```
import java.awt.Point;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public aspect CreatorImplementation {
   private static Point Creator.lastFrameLocation = new Point(0, 0);
   public final void Creator.showFrame() {
      //...
   }
}
```

6.4. Refactorings for Restructuring the Internals of Aspects

The refactorings from this group deal with the task of improving the internal structure of an aspect, after all elements from a crosscutting concern were moved into it (using the refactorings presented in the previous section). The group is headed by *Tidy Up Internal Aspect Structure* (128), a composite refactoring that provides the general framework for using the remaining refactorings. The motivation for *Tidy Up Internal Aspect Structure* (128) is due to the finding that aspects resulting from feature extractions are generally badly formed, betraying much duplication and inadequate internal structure. There is duplication because programmers are forced to duplicate code in multiple classes, since they cannot modularise the related concern. The act of moving these duplicated elements does not eliminate the duplication; it merely moves it to a single place, where it is clearly exposed. It is usual for extracted aspects to excessively rely on inter-type declarations (because with them it is easy to move members from the base code). When all the members are within the aspect, it is time to check whether they should stay as they are, or they should be replaced by a different mechanism.

Our experiments demonstrated that applying *Generalise Target Type with Marker Interface* (113) is an effective way to expose and remove various kinds of duplication in multiple inter-type declarations of identical members. We also recommend it as a starting step to tidying up the internal structure of aspects resulting from extractions. When using *Generalise Target Type with Marker Interface* (113) it may be expedient to use *Extend Marker Interface with Signature* (113) as a stopgap for some particular cases when the generally applicable code is being separated from case-specific code.

When you notice the *Aspect Laziness* (80) smell, use *Replace Inter-type Field with Aspect Map* (118) and *Replace Inter-type Method with Aspect Method* (125) to replace the introductions with a different logic providing similar functionality but also with the desired flexibility. *Introduce Aspect Protection* (116) is used to restore some protection to members that were turned public in the process of being moved to an aspect.

6.4.1 Extend Marker Interface with Signature

Typical situation

An inner interface represents a role used only within the aspect. You would like the aspect to call a method specific to a type implementing the interface, but not declared by it.

Recommended action

Add in the aspect an inter-type abstract declaration of the needed case-specific signature, targeting the interface.

Motivation

Sometimes you would like to temporarily resolve a dependence on case-specific part because that would enable you to do some tidying up of the aspect's internals, after which you will be in a better position to deal with the dependence. *Extend Marker Interface with Signature* can be used as a stopgap in such situations to temporarily resolve dependences to a type-specific method. One case in which this situation arises often is during the use of *Generalise Target Type with Marker Interface* (113).

An alternative solution to these problems would be to resort to downcasts. However, downcasts create dependencies to the target type of the cast: the specific type will need to be included in the aspect's import section, the type's binary file will have to be available when performing a build, etc. *Extend Marker Interface with Signature* can be preferable in some situations because it avoids such dependencies. The dependence it creates is restricted to a method signature only, not to specific types. For these reasons, this refactoring may be worth using even in (simple) cases.

Preconditions

The signature must be public in order to be acceptable to the compiler. In addition, this solution is feasible only if all the types made to implement the marker interface export the signature.

Mechanics

- If the method is not public, change it to public.
- Create in the aspect an inter-type abstract declaration of the method's signature targeting the marker interface that will be used in place of the specific type.
- Compile and test.

Example

The ExampleAspect aspect uses the Role marker interface. Some instructions using Role resort to a downcast to specific type SpecificType, to resolve the call to the doSomething method, which is specific to this type. By using *Extend Marker Interface with Signature,* we eliminate this dependence to SpecificType. If this is the only use of SpecificType within ExampleAspect, even the import can be removed.

```
import ...SpecificType;

public aspect ExampleAspect {
   private interface Role { }
   ... action(Role obj) {
      //...
      ((SpecificType)obj).doSomething()
```

```
import ...SpecificType;

public aspect ExampleAspect {
   private interface Role { }
   public abstract void Role.doSomething();
   //...
   obj.doSomething()
```

6.4.2 Generalise Target Type with Marker Interface

Typical situation

An aspect refers to specific concrete types, preventing it from being reused.

Recommended action

Replace the references to specific types with a marker interface and make the specific types implement the marker interface.

Motivation

This refactoring contributes to reduce the coupling between an aspect and its target code bases. It can also be used to expose and eliminate much duplication that couldn't be eliminated if the code kept referring to specific types. It can also be useful when we want to apply *Extract Superaspect* (129) to aspects containing providing similar functionality, because it contributes to rationalise its internal structures.

Several situations can prevent *Extract Superaspect* (129) from being immediately applied. The concrete aspects can contain code specific to concrete classes in the midst of generally applicable code. If a general marker interface could be used instead of the specific types, use *Generalise Target Type with Marker Interface*. After applying this refactoring, the resulting marker interfaces may be candidates for pulling up to a superaspect.

Mechanics

- Create a marker interface representing the role played by the target classes. Create the 'declare parents' to associate the concrete classes to the role.
- Replace the references to the class with references to the marker interface. In cases when the aspect introduces the same field or method to more than one class, remove the duplication by replacing the various introductions with a single introduction to the interface.
  Sometimes the replacement cannot be made in method bodies because parts of the code depend on elements specific to a concrete class. In such cases, consider using *Extract Method* ([47], p.110) to separate the parts covered by the role interface from the parts specific to particular classes. This may be an indication that in future the aspect should be split into a generally applicable abstract superaspect and one or several specific concrete subaspects, using *Extract Superaspect* (129).
- Compile and test.

- When all method introductions refer to the interface, it is possible to remove the declarations of operations (methods) within the interface (if the interface is a inner interface, nested within the aspect, the related operations are defined within the aspect anyway, so removing the declarations from the interface will result in simpler code). If, however, the interface is kept standalone, leave the declarations in place. This way the code will be easier to understand.

Example: Simple Replacements

In the following example, GUIColleague is an interface, representing a role. The aspect Mediator assigns the GUIColleague role to the Button class, but some parts of the code still specifically refer to Button instead of GUIColleague. We want to make all code to depend only on the interface.

```
public aspect Mediator {
   declare parents: Button implements GUIColleague;
   declare parents: Label implements GUIMediator;
   GUIMediator Button._mediator;
   public void Button.setMediator(GUIMediator mediator) {
      this._mediator = mediator;
   }
   pointcut buttonClicked(Button button):
      execution(public void clicked()) && this(button);
   after(Button button): buttonClicked(button) {
      button._mediator.colleagueChanged(button);
   }
   //...
}
```

```
public aspect Mediator {
   declare parents: Button implements GUIColleague;
   declare parents: Label implements GUIMediator;
   GUIMediator GUIColleague._mediator;
   public void GUIColleague.setMediator(GUIMediator mediator) {
      this._mediator = mediator;
   }
   pointcut buttonClicked(GUIColleague button):
      execution(public void clicked()) && this(button);
   after(GUIColleague button): buttonClicked(button) {
      button._mediator.colleagueChanged(button);
   }
   //...
}
```

Naturally, the names of some variables (such as button) should now be renamed to reflect their more general context.

Example: Eliminating Duplication

This example is based on the Observer pattern ([48], pp.293-303). See also section 7.1 for more information on Observer. In this example, the ObservingOpen aspect encapsulates an observing relationship that was extracted from the participant classes. ObservingOpen introduces some fields and methods into several classes playing the Observer role (in this case Bee and Hummingbird). The classes are the only difference between these introductions, so applying *Generalise Target Type with Marker Interface* creates the Subject marker interface and removes the duplication.

```
public aspect ObservingOpen ... {
   //...
   private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
   private OpenObserver Bee.openObsrv = new OpenObserver(this);

   public java.util.Observer Bee.openObserver() {
      return openObsrv;
   }
   public java.util.Observer Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

```
public aspect ObservingOpen ... {
   //...
   private interface Subject { }
   declare parents: (Bee || Hummingbird) implements Subject;
   private OpenObserver Subject.openObsrv = new OpenObserver(this);

   public java.util.Observer Subject.openObserver() {
      return openObsrv;
   }
}
```

6.4.3 Introduce Aspect Protection

Typical situation

You would like an inter-type member to be visible in an aspect al all its subaspects, but not outside the aspect inheritance chain.

Recommended action

Declare the inter-type member as public and place a 'declare error' preventing its use outside the aspect inheritance chain.

Motivation

AspectJ does not allow the protected access on inter-type members, so whenever we would like to extend its access to subaspects we must classify the member as public. In some cases, it is desirable to have some form of access protection preventing the use of the member outside aspect code. The 'declare error' mechanism enables us to emulate that protection.

Mechanics

- Add a 'declare warning' in the aspect enclosing the inter-type member, specifying the intended restriction on its use.
- Compile and test.
- For every warning generated by the compiler, perform the refactorings necessary to move the member to the authorised modules of the system.
- When there are no more warnings change the 'declare warning' to 'declare error'.

Example: protecting an inter-type field

Consider an abstract superaspect GeneralPolicy declaring inter-type the field _sensitiveData. We want to restrict use of the field to aspect and its subaspects.

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   //...
}
```

```
aspect ConcretePolicy extends GeneralPolicy {
   //code using Participant._sensitiveData
}
```

We can add in the superaspect the following 'declare warning':

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   declare warning:
      (set(public Data Participant+._sensitiveData) ||
      get(public Data Participant+._sensitiveData))
      && !within(GeneralPolicy+):
      "field _sensitiveData is aspect protected. Not visible here.";
   //...
}
```

Next, we deal with all points in the system giving rise to warnings. After all warnings are gone, we change the 'declare warning' to 'declare error'.

Example: protecting an inter-type method

Suppose the same abstract aspect as in the previous example also includes method processSensitiveData, which we also would like to protect:

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   public void processSensitiveData() {
      //code using Participant._sensitiveData
   }
   //...
}
```

We create the following 'declare warning':

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   public void processSensitiveData() {
      //code using caspule._sensitiveData
   }
   declare warning:
      call(void processSensitiveData())
      && !within(GeneralPolicy+):
      "method processSensitiveData is aspect protected. Not visible here.";
   //...
}
```

Likewise, the 'declare warning' should be changed to 'declare error' when all the warnings are gone.

Example: protecting an inter-type method relative to both the host class and the aspect chains

What if we want to allow the access to a member in the host class, in addition the aspect and their descendents? In the above example all is needed is one more within to the above 'declare error':

```
declare error:
   call(void processSensitiveData())
   && !within(Participant+)
   && !within(GeneralPolicy+):
   "Call to processSensitiveData() outside Participant and GeneralPolicy
chains.";
```

6.4.4 Replace Inter-type Field with Aspect Map

Typical situation

An aspect statically introduces additional state to a set of classes, when a more dynamic or flexible link between state and targets would be desirable.

Recommended action

Replace the inter-type declarations with a structure owned by the aspect performing a map between the additional state and target objects.

Motivation

An inter-type declaration is a static mechanism. It affects all instances of the target class, throughout their entire life cycles. For some problems, this is exactly right, but for others something more flexible would be preferable. In some cases only a subset of all instances of a class need the extra state and behaviour, or they need it only in a specific phase of their life cycles. Sometimes the same instance simultaneously needs multiple instances of the extra state and behaviour. Sometimes the application only knows at runtime which instances need the extra state and behaviour. Inter-type declarations do not provide the necessary flexibility in these cases.

An inter-type declaration is itself a kind of mapping, usually from a class to a field or method. The problem is that we cannot control the moments when it applies, when it ceases to apply, and the precise set of objects to which it applies. Whenever this kind of flexibility is required and the existing solution relies on introductions, use *Replace Inter-type Field with Aspect Map* to replace the introductions with a suitable mapping.

This refactoring is also useful in a different situation. Sometimes we have several aspects performing similar actions on similar data, and these include inter-type declarations. Naturally, we'll want to remove the duplication, by pulling the common parts to a superaspect. Here arises another problem: as long as each subaspect introduces its own additional fields to classes, there will be separate instances of the additional state for each subaspect. However, if the code is pulled up to the superaspect, there will be a single instance of the introduced state common to all subaspects. A similar problem would arise if we tried to replace an instance field with a static field. Such a pull will almost certainly not be behaviour preserving. In most cases, an inter-type declaration cannot be pulled up to a superaspect as is. This pulling up usually requires the prior replacement of inter-type state with aspect state.

As it happens, the kind of replacements that solve the first problem can solve the second problem as well. Unlike with inter-type declarations, there is a separate instance of the state declared in the superaspect in each active subaspect. In most cases, solving the problem merely entails selecting a suitable structure to replace the inter-type fields, and update the associated logic accordingly.

To ease the replacement of the original inter-type state with the new mapping structure, you should first isolate it behind a small layer within the aspect, to protect the rest of the aspect code from being exposed to it. In the simplest case, all we have to do is ensure that the aspect is provided with accessor methods encapsulating the inter-type fields. Only

those methods will need to be changed when the structure is replaced. In the case of preparing inter-type declarations to be pulled up, *Replace Inter-type Field with Aspect Map* must be applied to each of subaspects in turn. Next, use *Pull Up Field* ([47], p.320) and *Pull Up Method* ([47], p.322) to pull the state and its associated logic to the common superaspect.

Preconditions

Ensure that the fields in the various aspects do indeed provide equivalent interfaces and functionality.

Mechanics

- Use *Encapsulate Field* ([47], p.206) on the introduced field. Unlike traditional accessor methods, these ones are aspect methods, receiving the target object as argument.
- Add to each aspect a new structure capable of supporting the equivalent functionality. Add accessors similar to the ones created in the previous step: they would ideally have the same signatures and similar names. Add any additional management methods (i.e. for insertion, removal, etc.) that may also be required.
- If the aspect resorts to inter-type methods to handle the field, use *Replace Inter-type Method with Aspect Method* (125) to create aspect versions of those methods, using the new structure.
- Compile and test.
- Replace each call to the original accessors with the new ones. Compile and test when all replacements are done.
- Remove the old accessor methods. Compile and test.
- Remove the old inter-type field and related code. Compile and test.

Example: replacing an inter-type field with an aspect map

The following example presents fragments of an aspect implementing an instance of the Mediator pattern ([48], pp.273-282), adapted from a Java implementation by Cooper [32]. In this example, there is a mediator object (of type Mediator) acting as the hub of communication between various colleagues. The colleagues are instances of ClearButton and MoveButton, both subclasses of javax.swing.JButton, and KidList, which is a subclass of javax.swing.JScrollPane, implementing a listener interface from the javax.swing.event API. This example declares the Colleague role as a marker interface and assigns it to the three colleague participant types. The aspect indirectly introduces in each colleague a reference to the mediator, by way of the marker interface.

This implementation is unsuitable because it introduces the additional state and behaviour to all instances of the participant classes, independently of whether all of them need it or not. By replacing this implementation with one based on a map, we eliminate this inflexibility.

```
public aspect Mediating ...
   private interface Colleague {}
   private Mediator Colleague.mediator;

   declare parents:
      (ClearButton || MoveButton || KidList) implements Colleague;

   pointcut clearButtonExecute(ClearButton clearButton): ...
   after(ClearButton clearButton): clearButtonExecute(clearButton) {
      clearButton.mediator.clear();
   }

   pointcut moveButtonExecute(MoveButton moveButton): ...
   after(MoveButton moveButton): moveButtonExecute(moveButton) {
      moveButton.mediator.move();
   }

   pointcut kidListChanged(KidList kidList): ...
   after(KidList kidList) returning: kidListChanged(kidList) {
      kidList.mediator.select();
   }
```

As a first step, we perform a refactoring similar of *Encapsulate Field* ([47], p.206) to produce a temporary getter method for the inter-type field. The same getter can be used in all different target types. It cannot be given exactly the same name as the final getter, so we add a zero to avoid compiler errors.

```
Public aspect Mediating ...
   private Mediator getMediator0(Colleague colleague) {
      return colleague.mediator;
   }
   pointcut ...
   after(ClearButton clearButton): clearButtonExecute(clearButton) {
      getMediator0(clearButton).clear();
   }
   pointcut ...
   after(MoveButton moveButton): moveButtonExecute(moveButton) {
      getMediator0(moveButton).move();
   }
   pointcut ...
   after(KidList kidList) returning: kidListChanged(kidList) {
      getMediator0(kidList).select();
   }
```

Now that all accesses to the inter-type field are done through this temporary getter, the inter-type nature of the mediator field was encapsulated. Next, we add a suitable data structure to map the target objects to the mediator field. A hash table is a good choice for these cases. The introduced field was private to the aspect, so the getters are private as well. The access of the final setter can be more problematic. Note that the final setter is responsible for associating the target object with the mediator field, using the newly added mapping structure. The final setter does not have a correspondent in the old version of the code. Nevertheless, it must be called in the appropriate point of the program. The access of the final setter depends on where the field is used in the system: private if it is used only within the aspect, non-private otherwise. In this example, we assume a public access.

```
import java.util.WeakHashMap;

public aspect Mediating ...
   WeakHashMap colleague2mediatorMap = new WeakHashMap();

   private Mediator getMediator(Colleague colleague) {
      return (Mediator)colleague2mediatorMap.get(colleague);
   }
   public void setMediator(Colleague colleague, Mediator mediator) {
      colleague2mediatorMap.put(colleague, mediator);
   }
   private Mediator getMediator0(Colleague colleague) {
      return colleague.mediator;
   }
```

We now add the calls to the final setter in the client code. The places where the objects containing the field are created could be used as a basis, though in some cases it may be preferable to place the calls elsewhere. After all, that is precisely one of the advantages of replacing a static mapping with a dynamic one: we have more choices. Outside the aspect, the calls to the final setter should be something like this:

```
Mediating.aspectOf().setMediator(clearButton, mediator);
```

Inside advice within the aspect, the same call can be expressed in a simpler way:

```
setMediator(clearButton, mediator);
```

We insert the calls to the final setter and make the calls to the temporary getter refer to the final getter. After compiling and testing again, we can delete the original declaration and the temporary getter. Now the aspect's code looks like this:

```
public aspect Mediating ...
   private Mediator Colleague.mediator;
   declare parents: (ClearButton || MoveButton || KidList)
      implements Colleague;

   WeakHashMap colleague2mediatorMap = new WeakHashMap();

   private Mediator getMediator(Colleague colleague) {
      return (Mediator)colleague2mediatorMap.get(colleague);
   }
   public void setMediator(Colleague colleague, Mediator mediator) {
      colleague2mediatorMap.put(colleague, mediator);
   }

   private Mediator getMediator0(Colleague colleague) {
      return colleague.mediator;
   }
   pointcut clearButtonExecute(ClearButton clearButton): ...
   after(ClearButton clearButton): clearButtonExecute(clearButton) {
      getMediator(clearButton).clear();
   }
   pointcut moveButtonExecute(MoveButton moveButton): ...
   after(MoveButton moveButton): moveButtonExecute(moveButton) {
      getMediator(moveButton).move();
   }
   pointcut kidListChanged(KidList kidList): ...
   after(KidList kidList) returning: kidListChanged(kidList) {
      getMediator(kidList).select();
   }
```

Example: preparing an Observer implementation for the extraction of a superaspect

This second example is an implementation of the Observer pattern ([48], pp.293-303). See also section 7.1 for more information on Observer. This implementation was extracted into an aspect from the example by Cooper [32], using *Extract Feature into Aspect* (90). This example is a bit more complex than the previous one, because it includes inter-type methods that use the inter-type field. These inter-type methods must be replaced using *Replace Inter-type Method with Aspect Method* (125). We assume the scenario in which the system has other, similar, implementations of the pattern and we would like to factor out the common elements by pulling them up to a superaspect. These implementations rely on the introduction of a java.util.Vector field to the subject participant, which is among the elements we would like to pull up, along with its associated logic.

The present implementation does not lend itself to be pulled up to the superaspect, for the same reasons as in the previous example: it was designed assuming there would be only one instance of the pattern for each subject: the vector cannot support multiple observing relationships for the same object. To solve this problem, we'll replace the inter-type vector with a more suitable hash table owned by the aspect, which will manage the mappings between subjects and the list (i.e. a java.util.Vector object) of its observers. We'll use *Replace Inter-type Method with Aspect Method* (125) to replace the original logic using the vector with aspect logic using the hash table.

Cooper's example includes a Watch2LSubject object as subject and two types of observers, which are instances of ListFrameObserver and ColorFrameObserver (both subclasses of javax.swing.JFrame). The Watch2LSubject object includes three radio buttons, one for each of the colours red, green and blue. Whenever a different radio button is selected, the ColorFrameObserver instances change their background colour accordingly, and the ListFrameObserver adds the name of the selected colour to its list.

The refactored aspect uses two inner interfaces[27] to represent the roles of subject and observer. It introduces the java.util.Vector field to the objects playing the role of subject, which holds the subject's registered observers. The aspect also introduces two methods to the subjects: addObserver(Observer), which is used to register a new observer for the subject, and notifyObservers(JRadioButton), through which subjects notify all their registered observers of a change in the selected colour. That notification is carried out through the sendNotify method, which is declared in the Observer inner interface. The sendNotify method receives as parameter a string representing the new colour. The aspect also introduces the implementation of sendNotify for each concrete observer type.

[27] In the initial Java implementation [32], Observer and Subject were standalone interfaces. They were inlined to within the Observing aspect during the refactoring process.

```
public aspect Observing ...
   private interface Subject {}
   interface Observer {
      /** notify the Observers that a change has taken place */
      public void sendNotify(String s);
   }
   declare parents: Watch2LSubject implements Subject;
   declare parents: (ListFrameObserver || ColorFrameObserver)
      implements Observer;

   private Vector Subject._observingFramesList = new Vector();

   public void Subject.addObserver(Observer obs) {
      //   adds observer to list in Vector
      _observingFramesList.addElement(obs);
   }
   /* sends text of selected button to all observers */
   private void Subject.notifyObservers(JRadioButton rad) {
      String sColor = rad.getText();
      for (int i = 0; i < _observingFramesList.size(); i++ ) {
         ((Observer) (_observingFramesList.elementAt(i))).sendNotify(sColor);
      }
   }

   public void ListFrameObserver.sendNotify(String s) {
      _listData.addElement(s);
   }
   public void ColorFrameObserver.sendNotify(String str) {
      changeColor(str);
   }
```

The aspect also includes a pointcut and corresponding advice to trigger the adequate behaviour when the subject changes the selected colour:

```
   pointcut watchStateChange(Watch2LSubject watch, ItemEvent event): ...
   after(Watch2LSubject watch, ItemEvent event):
         watchStateChange(watch, event) {
      if(event.getStateChange() == ItemEvent.SELECTED)
         watch.notifyObservers((JRadioButton) event.getSource());
   }
```

The mechanics prescribe the use of *Encapsulate Field* ([47], p.206) on the existing field. In this particular case, we must instead create a new field as the mapping structure (we'll create the accessor methods for the structure as soon as there is a need to do so).

```
import java.util.WeakHashMap;
...
public aspect Observing ...
   //...
   WeakHashMap _subject2Observers = new WeakHashMap();
```

Next, we use *Replace Inter-type Method with Aspect Method* (125) to replace the addObserver and notifyObservers inter-type methods with aspect versions using the new mapping structure (see the example section of *Replace Inter-type Method with Aspect Method* (125) for more details of this step).

The new implementation is now in place and working. There was no need to add accessors to the mapping structure, as it is already encapsulated by addObserver and notifyObservers. These two aspect methods comprise a small layer hiding the structure. We can now delete the old implementation, after which the aspect looks like this:

```
public aspect Observing ...
   private interface Subject {}
   interface Observer {
      /** notify the Observers that a change has taken place */
      public void sendNotify(String s);
   }
   declare parents: Watch2LSubject implements Subject;
   declare parents: (ListFrameObserver || ColorFrameObserver)
      implements Observer;

   private Vector Subject._observingFramesList = new Vector();

   public void Subject.addObserver(Observer obs) {
      //   adds observer to list in Vector
      _observingFramesList.addElement(obs);
   }
   /* sends text of selected button to all observers */
   private void Subject.notifyObservers(JRadioButton rad) {
      String sColor = rad.getText();
      for (int i = 0; i < _observingFramesList.size(); i++ ) {
         ((Observer) (_observingFramesList.elementAt(i))).sendNotify(sColor);
      }
   }

   WeakHashMap _subject2Observers = new WeakHashMap();

   public void addObserver(Subject subject, Observer observer) {
      Vector observers;
      Object obj = _subject2Observers.get(subject);
      if(obj == null)
         observers = new Vector();
      else observers = (Vector) obj;
      observers.add(observer);
      _subject2Observers.put(subject, observers);
   }
   public void notifyObservers(Subject subject, JRadioButton radioButton) {
      String sColor = radioButton.getText();
      Vector observersList = (Vector)_subject2Observers.get(subject);
      for (int i = 0; i < observersList.size(); i++ ) {
         ((Observer) (observersList.elementAt(i))).sendNotify(sColor);
      }
   }
   public void ListFrameObserver.sendNotify(String s) {
      _listData.addElement(s);
   }
   /* Observer is notified of change here */
   public void ColorFrameObserver.sendNotify(String str) {
      changeColor(str);
   }

   pointcut watchStateChange(Watch2LSubject watch, ItemEvent event): ...
   after(Watch2LSubject watch, ItemEvent event):
        watchStateChange(watch, event) {
      if(event.getStateChange() == ItemEvent.SELECTED)
         notifyObservers(watch, (JRadioButton) event.getSource());
   }
}
```

6.4.5 Replace Inter-type Method with Aspect Method

Typical situation

An aspect introduces additional methods to a class or interface, when a more dynamic and flexible composition would be desirable.

Recommended action

Replace the inter-type method with an aspect method getting the target object as parameter.

Motivation

This refactoring was designed to be used as a follow-up to *Replace Inter-type Field with Aspect Map* (118). That refactoring deals with inter-type fields and the present refactoring deals with the (inter-type) methods that use those fields.

The present refactoring is made possible by the fact that a method introduced to a class can always be replaced by a similar aspect method receiving an instance of the target class as an additional argument, which will use the target object as a key.

```
public class Capsule {
   private int _value;
   public Capsule(int value) {
      _value = value;
   }
```

```
public aspect Additional {
   public void Capsule.doSomethingMore() {
      System.out.println("Doing something more with capsule" + this);
   }
```

```
      Capsule capsule = new Capsule(7);
      capsule.doSomethingMore();
```

```
public class Capsule {
   private int _value;
   public Capsule(int value) {
      _value = value;
   }
```

```
public aspect Additional {
   public void doSomethingMore(Capsule capsule) {
      System.out.println("Doing something more with " + capsule);
   }
```

```
      Capsule capsule = new Capsule(7);
      Additional.aspectOf().doSomethingMore(capsule);
```

Replacements of this kind should not be made in the general case, and that is why we prescribe using this refactoring only in the context of *Replace Inter-type Field with Aspect Map* (118). This refactoring is equally useful to deal with both situations covered by the other refactoring: (1) replacing inter-type declarations with a dynamic mechanism and (2) preparing inter-type state duplicated in various aspects to be factored out to a common superaspect. This refactoring transforms existing inter-type methods into aspect methods based on the map that was created when applying *Replace Inter-type Field with Aspect Map* (118).

Mechanics

- Create in the aspect a copy of the inter-type method, with same name and signature. Insert, in the beginning of the aspect method's parameter list, an additional parameter whose type is the original target of the inter-type declaration.

- Replace each reference to 'this' with the new parameter. Change all self-calls and references to fields to refer to the new first parameter.
- Compile and test.
- Change the body of the inter-type method so that it calls the aspect method, if there are no further dependences preventing you.
- Add a 'declare warning' exposing all calls to the inter-type method:

```
declare warning:
    (call(<type> <host class>.someMethod(<arguments>)):
    "method <host class>.someMethod() is called here.";
```

- Following the warnings, replace each call to the inter-type method with a call to the aspect method. Compile and test after each change.
- When there are no more warnings delete the 'declare warning' and the inter-type method.
  When covering the mechanics of several refactorings from [47], Fowler considers the situation when the existing method is part of the interface and cannot be changed. Fowler recommends that in such cases the old method be left in place and marked as deprecated.
- Compile and test.

Example

This example is part of the second example for *Replace Inter-type Field with Aspect Map* (118). In it, an aspect introduces the following methods to the Subject marker interface:

```
public void Subject.addObserver(Observer obs) {
   _observingFramesList.addElement(obs);
}
private void Subject.notifyObservers(JRadioButton rad) {
   String sColor = rad.getText();
   for (int i = 0; i < _observingFramesList.size(); i++ ) {
      ((Observer) (_observingFramesList.elementAt(i))).sendNotify(sColor);
   }
}
```

As an example of client code, the following subject and observers are created and registered, through calls to the Subject.addObserver method:

```
Watch2LSubject subject = new Watch2LSubject();
//Observing.aspectOf().setSubject(subject);

ColorFrameObserver cframeObs1 = new ColorFrameObserver();
ColorFrameObserver cframeObs2 = new ColorFrameObserver();
ColorFrameObserver cframeObs3 = new ColorFrameObserver();
ListFrameObserver lframeObs = new ListFrameObserver();

subject.addObserver(cframeObs1);
subject.addObserver(cframeObs2);
subject.addObserver(cframeObs3);
subject.addObserver(lframeObs);
```

The aspect itself also includes an advice calling the other method, Subject.notifyObservers:

```
after(Watch2LSubject watch, ItemEvent event):
     watchStateChange(watch, event) {
   if(event.getStateChange() == ItemEvent.SELECTED)
      watch.notifyObservers((JRadioButton) event.getSource());
}
```

This functionality should be replaced by aspect methods based on a hash table owned by the aspect: the aspect field _subject2Observers, which uses subject objects as keys, and vectors of observers as values:

```
WeakHashMap _subject2Observers = new WeakHashMap();
```

As a first step, we create the following two aspect methods, with the same names:

```
public void addObserver(Subject subject, Observer observer) {
   Vector observers;
   Object obj = _subject2Observers.get(subject);
   if(obj == null)
      observers = new Vector();
   else observers = (Vector) obj;
   observers.add(observer);
   _subject2Observers.put(subject, observers);
}
public void notifyObservers(Subject subject, JRadioButton radioButton) {
   String sColor = radioButton.getText();
   Vector observersList = (Vector)_subject2Observers.get(subject);
   for (int i = 0; i < observersList.size(); i++ ) {
      ((Observer) (observersList.elementAt(i))).sendNotify(sColor);
   }
}
```

We can't replace the body of the inter-type methods with calls to the new ones at this point. We must first replace the calls to the addObserver method, which register the observers to their subjects. Otherwise, the tests would fail. We therefore perform the next step as prescribed, adding 'declare warning' clauses that will expose all calls to these methods:

```
declare warning: call(void Subject.addObserver(Observer)):
   "Method Subject.addObserver(Observer) is called here.";
declare warning: call(void Subject.notifyObservers(JRadioButton)):
   "Method Subject.notifyObservers(JRadioButton) is called here.";
```

We compile, resulting in a series of warnings locating the calls to the old methods. After replacing each of them with calls to the aspect methods, we compile again. All warnings disappeared, and we test. We remove the 'declare warning' clauses. Now the client code calling addObservers looks like this:

```
   Watch2LSubject watch2LFrame = new Watch2LSubject();

   ColorFrameObserver cframeObs1 = new ColorFrameObserver();
   ColorFrameObserver cframeObs2 = new ColorFrameObserver();
   ColorFrameObserver cframeObs3 = new ColorFrameObserver();
   ListFrameObserver lframeObs = new ListFrameObserver();

   subject.addObserver(cframeObs1);
   subject.addObserver(cframeObs2);
   subject.addObserver(cframeObs3);
   subject.addObserver(lframeObs);

   Observing.aspectOf().addObserver(watch2LFrame, cframeObs1);
   Observing.aspectOf().addObserver(watch2LFrame, cframeObs2);
   Observing.aspectOf().addObserver(watch2LFrame, cframeObs3);
   Observing.aspectOf().addObserver(watch2LFrame, lframeObs);
```

The call to notifyObservers now takes the form:

```
after(Watch2LSubject watch, ItemEvent event):
      watchStateChange(watch, event) {
   if(event.getStateChange() == ItemEvent.SELECTED)
      notifyObservers(watch, (JRadioButton) event.getSource());
}
```

6.4.6 Tidy Up Internal Aspect Structure

Typical situation

The internal structure of an aspect resulting from the extraction of a crosscutting concern is sub-optimal, being based on static compositions and betraying duplication.

Recommended action

Tidy up the internal structure of the aspect by removing duplication and dependencies on case specific target types.

Motivation

This refactoring serves as the general framework indicating when to use the remaining refactorings from the same group[28], and in what situations.

AOP adds a new type of situation in which code duplication can arise (i.e. is exposed). Refactoring an object-oriented (OO) code base to aspects entails extracting concerns and features whose very crosscutting nature gives rise to duplication that is hard or impossible to avoid when using traditional OO mechanisms. A typical situation is a system containing repeated implementations of the same functionality scattered in multiple classes. Simply extracting those code snippets into an aspect does not guarantee, by itself, removal of this duplication. This merely moves the duplicated code into aspects. In some cases, the duplication becomes obvious only when it is placed in a single module. Therefore, extracting the crosscutting code is only the first part of the job. Next, duplication within the aspect must be removed and its internal structure improved. The necessary refactorings may involve profound changes in that structure, and may entail the replacement of the extracted design with a different, more suitable design.

Inter-type declarations can sometimes be too limiting. They make it very easy to move members from classes to aspects without impact on client code, and aspects resulting from extractions are likely to use them. However, in some cases, we would like the aspect to introduce the additional state and behaviour on an object-by-object basis, and inter-type declarations are not flexible enough to achieve that. This entails the replacement of these introductions with different logic.

Mechanics

- If the code assigns roles to participant classes, see if the aspect code uses marker interfaces to represent those roles instead of referring directly to case-specific classes. If it is not the case, use *Generalise Target Type with Marker Interface* (113).
- If parts of the code make explicit references to specific classes that cannot be generalised, separate the specific parts from the generally applicable ones by using *Extract Method* ([47], p.110). You should do this if the aspect contains enough generally applicable logic to be worth extracting to a reusable abstract superaspect.
- Inspect the inter-type declarations looking for cases in which the extra state and behaviour is needed only at specific times, or is needed by only a subset of the instances of the target classes, or may be needed in multiple instances simultaneously. In such cases, consider using *Replace Inter-type Field with Aspect Map* (118) to deal with the introduced state, and *Replace Inter-type Method with Aspect Method* (125) to deal with the behaviour based on that state.

[28] We do not mean to imply that each of the refactorings from te group is necessarily referred *directly*.

Example

The refactoring process described in subsection 7.7.2 is a good example of this composite refactoring.

6.5. Refactorings to Deal with Aspect Generalisation

We sometimes find that an aspect contains both generically applicable code and case-specific code. The former can potentially be placed in a reusable superaspect, in which case we use *Extract Superaspect* (129). That refactoring in turn prescribes the various *Pull Up* refactorings. Our experience showed that we sometimes need to reverse some of the pulls, and that was our motivation for the *Push Down* refactorings.

The main refactoring for this group is *Extract Superaspect* (129). This group bears the same name as an equivalent group in [47], which contains various *Pull Up* and *Push Down* refactorings. This section contains similar refactorings dealing in this case with aspect-specific constructs, including pointcuts, advice, marker interfaces and inter-type declarations.

6.5.1 Extract Superaspect

Typical situation

Two or more aspects contain similar code and functionality.

Recommended action

Move the common features to a superaspect.

Motivation

It has been noted numerous times that common patterns sometimes only surface during development [69][42]. When a common pattern in the code is identified, the sensible thing to do is make that commonality clear, by creating a separate unit of modularity with an appropriate name and placing the common code there. An obvious benefit is the removal of duplication.

When we extract various concerns from a code base, we may later conclude that several of the resulting aspects are in fact different instances of a common pattern. Different aspects may turn out to be variant implementations of the same kind of functionality, but sometimes the similarities only become noticeable when we are able to see all the code of each concern in one place (commonalities among aspects should also be easier to notice after applying *Tidy up Internal Aspect Structure* (128) to each of them).

When such commonalities are identified, it is necessary to prepare the internal structure of the aspects so that the common parts are amenable to being extracted into a common superaspect. In the case of inter-type declarations, the needed action usually entails the replacement of the existing introductions with a different implementation, as suggested in *Replace Inter-type Field with Aspect Map* (118) and *Replace Inter-type Method with Aspect Method* (125).

Mechanics

- Create an empty superaspect. Make the concrete aspects inherit from the abstract aspect.

- One by one, use *Pull Up Marker Interface* (132), *Pull Up Field* ([47], p.320), *Pull Up Method* ([47], p.322), *Pull Up Pointcut* (133) and *Pull Up Advice* (130) to move common elements to the superaspect.

  It's usually easier to move the marker interfaces first.
  Pull Up Field ([47], p.320) and *Pull Up Method* ([47], p.322) cannot be used with inter-type declarations, because there will be only one instance of the introduced fields for all subaspects. When you want to pull up inter-type fields, consider applying *Replace Inter-type Field with Aspect Map* (118) to the inter-type fields and *Replace Inter-type Method with Aspect Method* (125) to the inter-type methods using those fields.
  If you find a signature common to all subaspects but with different logic, add an abstract declaration or default definition in the superaspect.

- Compile and test after each pull.

6.5.2 Pull Up Advice

Typical situation

All subaspects use the same advice acting on a pointcut declared in the superaspect.

Recommended action

Move the advice to the superaspect.

Motivation

As with other kinds of pulls, this situation is likely to arise when a commonality is being extracted from various subaspects. At first sight, it may look that most of the considerations applying to *Pull Up Method* also apply here, but there are important differences. There is no polymorphism, and no overriding: all pieces of advice defined along the inheritance chain execute whenever one joinpoint is reached.

Preconditions

Keep in mind that a piece of advice declared in a superaspect will run as many times as there are concrete subaspects weaved into the system. If that is not what you want, leave the advice in the subaspects. This won't be a problem if at any given time only one concrete subaspect is weaved into the system.

If the common pointcut over which the advice executes is still duplicated in the various subaspects, use *Pull Up Pointcut* (133) first.

If at least one of the subaspects use a piece of advice different from the others you should not pull up the advice, since pieces of advice cannot be overridden.

Mechanics

- Copy the body of the advice in the superaspect.
- Delete the advice in each of the subaspects.
- Compile and test.

6.5.3 Pull Up Declare Parents

Typical situation

All subaspects use the same 'declare parents'.

Recommended action

Move the 'declare parents' to the superaspect.

Motivation

This refactoring contributes to extract a reusable superaspect out of two or more subaspects that duplicate some logic.

Mechanics

- Ensure that all 'declare parents' assign the same roles to the same participants.
- Place a copy of the 'declare parents' in the superaspect.
- Delete the 'declare parents' in each of the subaspects.
- Compile and test.

6.5.4 Pull Up Inter-type Declaration

Typical situation

An inter-type declaration would be best placed in the superaspect.

Recommended action

Move the inter-type declaration to the superaspect.

Motivation

The pull up or push down of inter-type declarations presents issues not generally found in other aspect constructs. For this reason, the applicability of this refactoring is more restricted than for other *Pull Up* refactorings. The main motivation of other *Pulls* is the factoring out of commonality out of duplicated code. Occasionally it is also convenient to move an element to the superaspect even when there is no duplication, because we simply think it is best placed there. This refactoring applies to *only* this last case. The main reason why this is so, is the fact that the number of instances of the introduced member is different, depending on where the inter-type declaration is placed in the aspect inheritance chain.

Target objects (i.e. instances of classes affected by the inter-type declaration) will have one separate instance of the inter-type member for each subaspect. If the various inter-type declarations are factored out to a single declaration in a superaspect, the target objects will have a *single* instance of the introduced member. This situation is roughly similar to when an instance member is turned into a static member. Such a transformation is not likely to be behaviour preserving, particularly when applied to fields. Also, keep in mind that two or more inter-type declarations of the same member will conflict with each other if their scopes of visibility overlap. Consequently, if several subaspects declare the same member, those members must be private to the subaspects (this would present an additional hurdle, if the mechanics entailed first making them public).

Preconditions

In the general case, this refactoring can be used only when there is a single instance of the inter-type declaration to pull up. Otherwise, whenever inter-type members include both fields and methods, deal with the fields first, using *Replace Inter-type Field with Aspect Map* (118) as a follow-up (the inter-type methods may themselves be replaced by something else during that process). *Pull Up Inter-type Declaration* can also be applied to duplicated inter-type methods that do not refer to inter-type fields.

Mechanics

- Create a new inter-type declaration in the superclass. If the declaration is private, relax the access to public and use *Introduce Aspect Protection* (116).
- Delete the inter-type declaration in the subaspect.
- Compile and test.

6.5.5 Pull Up Marker Interface

Typical situation

All subaspects use a marker interface to model the same role.

Recommended action

Move the marker interfaces to the superaspect.

Motivation

This refactoring contributes to extract a reusable superaspect out of two or more concrete subaspects that duplicate some logic.

Preconditions

Marker interfaces do not generally need to declare signatures. If the interface is not blank, consider first refactoring the code in order to remove the signatures.

Mechanics

- Inspect all uses of the interfaces to ensure the roles they model are indeed the same. Ensure they have the same name.
- Create a new interface in the superaspect. If the interfaces are private, you will need to change them to protected.
- Delete the subaspect interfaces.
- Compile and test.

Example

```
public abstract aspect AbstractMediation {
   //...
}
```

```
public aspect MediationCase1 extends AbstractMediation {
   private interface Mediator { }
   //...
}
```

```
public aspect MediationCase2 extends AbstractMediation {
   private interface Mediator { }
   //...
}
```

```
public abstract aspect AbstractMediation {
   protected interface Mediator { }
   //...
}
```

```
public aspect MediationCase1 extends AbstractMediation {
   protected interface Mediator { }
   //...
}
```

```
public aspect MediationCase2 extends AbstractMediation {
   protected interface Mediator { }
   //...
}
```

6.5.6 Pull Up Pointcut

Typical situation

All subaspects declare identical pointcuts.

Recommended action

Move the pointcuts to the superaspect.

Motivation

The mechanics of this refactoring have strong similarities with that of *Pull Up Method* ([47], p.322).

Mechanics

- Inspect the pointcuts to ensure they are identical. If the pointcuts are similar but not identical, first see if both capture the same set of joinpoints. Next, change one of them into the other. If the pointcuts have different signatures, change the signatures to the one you intend to use in the superaspect.
  Often, the pointcuts relate to the same concept and have the same signature, but capture different sets of joinpoints in each specific case. In such cases keep the pointcuts in the subaspects and place an abstract declaration in the superaspect.
- Create a copy of the new pointcut in the superaspect. If the pointcuts are private or package protected classify the new pointcut as protected. If the pointcut uses other pointcuts that are present in the aspects but not in the superaspect, declare the corresponding abstract pointcuts on the superaspect.
- Delete the pointcuts in the subaspects.
- Compile and test.

6.5.7 Push Down Advice

Typical situation

A piece of advice is used by only some subaspects, or each subaspect requires a different advice.

Recommended action

Move the advice to the subaspects that use it.

Motivation

This situation can arise when we discover that a supposedly reusable piece of advice does not cover all cases after all. Since there is no polymorphism with advice, you cannot override the advice with a case-specific advice in the subaspect. In these cases, it is necessary to place a version of the advice in each subaspect.

Mechanics

- Copy the advice to each subaspect. Check for any imports that the subaspects may require resolving the advice's code.
- Remove the advice from the superaspect. Check for any imports that may no longer be needed.
- Compile and test.

Example

This example is taken from the complete refactoring process described in Chapter 7.

The following advice comes from ObserverProtocol, the reusable aspect for the Observer pattern created by Hannemann and Kiczales [60]:

```
public abstract aspect ObserverProtocol {
   //...
   after(Subject s): subjectChange(s) {
      Iterator iter = getObservers(s).iterator();
      while ( iter.hasNext() ) {
         updateObserver(s, ((Observer)iter.next()));
      }
   }
```

Suppose you want to implement a use case similar to Eckel's flower example of the same pattern [39], in which observers are notified only the first time a distinct event occurs, avoiding repeated notifications when the same event occurs without other events in between. We copy the advice to the subaspect and remove it from ObserverProtocol:

```
public abstract aspect ObserverProtocol {
   //...
```

```
public aspect ObservingOpen extends ObserverProtocol {
   //...
   after(Subject s): subjectChange(s) {
      Iterator iter = getObservers(s).iterator();
      while ( iter.hasNext() ) {
         updateObserver(s, ((Observer)iter.next()));
      }
   }
```

You can then add the intended functionality. Naturally, if you have several other subaspects of ObserverProtocol that are quite happy to use the former advice, it would be bad style to duplicate the advice in each of them. In such cases, it is preferable to parameterise the additional behaviour in the advice.

6.5.8 Push Down Declare Parents

Typical situation

A 'declare parents' in a superaspect is not relevant for all the subaspects.

Recommended action

Move the 'declare parents' to the subaspects where it is relevant.

Motivation

Push Down Declare Parents is the opposite of *Pull Up Declare Parents* (130). Declare parent clauses are more likely to be found in case-specific concrete subaspects than in abstract

superaspects. The concrete aspects are used to inherit some reusable logic from an abstract aspect, and the latter does not generally includes 'declare parents' clauses.

Mechanics

- Add the 'declare parents' in all subaspects that require it. Compile and test.
- Remove the 'declare parents' from the superaspect. Compile and test.

6.5.9 Push Down Inter-type Declaration

Typical situation

An inter-type declaration would be best placed in a subaspect.

Recommended action

Move the inter-type declaration to the subaspect where it is relevant.

Motivation

The pull up or push down of inter-type declarations presents issues not generally found in other aspect constructs. It is possible to declare the same member in multiple aspects, but their visibility scopes cannot overlap and therefore they must be private. In addition, pushing down an inter-type declaration entails replacing a single instance (per target object) of the member common to all subaspects with multiple instances (per target object), one for each subaspect. In the case of fields, this transformation is not behaviour preserving. The same applies to the opposite refactoring, *Pull Up Inter-type Declaration* (131). In the case of methods, *Push Down Declare Parents* risks leading to duplication.

Preconditions

For the above reasons, this refactoring should be used to push down inter-type declarations found in a *single* subaspect.

Mechanics

- Copy the declaration in the subaspect.
- Remove the declaration from the superaspect.
- Compile and test.

6.5.10 Push Down Marker Interface

Typical situation

A marker interface declared within a superaspect models a role used only in some subaspects.

Recommended action

Move the marker interface to those subaspects.

Motivation

Push Down Marker Interface is the opposite of *Pull Up Marker Interface* (132), and can be used to reverse its effects when they do not turn out as expected.

Preconditions

Inspect the superaspect to ensure it does not include any references to the marker interface, besides the declaration itself. If the interface is not private, also ensure no client code refers to the interface as belonging to the superaspect.

Mechanics

- Declare the marker interface in each of the superaspects.
- Delete the declaration from the superaspect.
- Compile and test.

6.5.11 Push Down Pointcut

Typical situation

A pointcut in the superaspect is not used by some subaspects inheriting it.

Recommended action

Move the pointcut to those subaspects that use it.

Motivation

A pointcut represents a set of interesting events that should trigger some response on the part of the aspects. Placing the pointcut in an abstract aspect leads programmers to assume that those events are interesting to any descendent of the abstract aspect. When that is not so, it can lead to confusion. If some of the aspects do not need the pointcut at all, this is an instance of *Refused Bequest* ([47], p.87) applied to pointcuts. As with the traditional instances of this smell, it is advisable to analyse the specific case to be sure this smell is worth cleaning.

In case only the concrete aspects know what are the interesting events, leave a protected abstract declaration of the pointcut in the superaspect. In addition, there are cases when the pointcut to be pushed down is used by advice or other pointcuts in the superaspect. These cases also require an abstract declaration of the pointcut.

Keep in mind that any inherited abstract declaration must be concretised in all concrete subaspects. If you really want to refuse the bequest you can override the inherited pointcut with one that doesn't captures any joinpoints, such as in the example below. However, this solution does not remove the *Refused Bequest* smell, and it would be best to use it only as a stopgap.

```
public abstract AbstractAspect {
   protected abstract pointcut refusedBequest(<parameters>);
```

```
public ConcreteAspect extends AbstractAspect {
   protected pointcut refusedBequest(<parameters>): !within(*);
```

Mechanics

- Declare the pointcut in all subaspects.
- Remove the pointcut from the superaspect. Leave an abstract declaration if one of the cases requiring one applies.
- Compile and test.

- If you did not leave an abstract declaration in the superaspect, remove the pointcut from any subaspect that does not need it.

6.6. Dealing with Legacy Code

The single refactoring covered in this section was specifically designed for dealing with legacy code with published interfaces[29]. By published interfaces, we mean APIs that are used by clients outside the control of the developer, meaning that the developer is not authorised or unable to change.

6.6.1 Partition Constructor Signature

Typical situation

A constructor initialises members related to a concern being extracted to an aspect. The initialisations use some of the constructor's parameters, which are not required when the concern being extracted is not present.

Recommended action

Create in the class a constructor devoid of any code relative to the extracted concern, including the arguments. Replace the code from original constructor that is not related to the extracted concern with a call to the new constructor. Move the original constructor to the aspect.

Motivation

This awkward situation arises when an interface is tangled with various concerns but for some reason cannot be changed, or we want to avoid our refactorings to impact on client code.

This refactoring cannot be applied if the constructor passes arguments to a super call. Consider simplifying the hierarchy in other ways, for instance resorting to one of the creational patterns proposed in [48].

Mechanics

- Create a new constructor in the class, with a shortened argument list, without the arguments related to the crosscutting concern.
- Move to the new constructor all the statements not related to the crosscutting concern.
- Place a call to this as the first statement in the original constructor, passing only the parameters not related to the crosscutting concern.
- Move the original, modified constructor to the aspect.
- Append '.new' after the original constructor's name in the aspect. For instance, suppose arg1 is not related to the crosscutting concern:

```
//In the aspect
public
SomeClass.new(Type1 arg1, Type2 arg2) {
    this(arg1);
}
```

[29] This refactoring was derived from the case study described in Chapter 3. The research relating to the specific problems of legacy code was not developed further.

- In case the aspect is placed in a separate package, check if the constructor's class is declared in the aspect's import section. Check also if all imports in the host class are still necessary: some may have been needed only for arguments moved to the aspect.
- Compile and test.

Example

```
public class TangledStack {
   //...
   public TangledStack(JFrame frame) {
      elements = new Object[S_SIZE];
      frame.getContentPane().add(_label);
      text.setText("[]");
      frame.getContentPane().add(_text);
   }
   //...
}
```

```
public class TangledStack {
   //...
   public TangledStack() {
      elements = new Object[S_SIZE];
   }
   //...
}
```

```
public aspect WindowView {
   //...
   public TangledStack.new(JFrame frame) {
      this();
      frame.getContentPane().add(_label);
      _text.setText("[]");
      frame.getContentPane().add(_text);
   }
   //...
}
```

A more complete example using this refactoring can be found in the example section of *Extract Feature into Aspect* (90).

Chapter 7. Validating Example: Refactoring Observer

This Chapter describes in detail a complete refactoring process using 17 of the refactorings proposed in the previous Chapter, serving as a first validation of the refactorings[30]. The example is based on a Java implementation of the Observer design pattern ([48], pp.293-303) by Bruce Eckel [39], which is refactored into a different implementation in AspectJ. From a certain point, the refactoring process is broken into two alternative paths. The first path is performed solely in terms of the original code. The second path takes advantage of the reusable aspect from the AspectJ implementation by Hannemann and Kiczales [60]. Both paths lead to solutions that broadly correspond to the same design.

Note that "refactoring" was here taken in the strict sense, with all its implications. These include the following:

- All code transformations are performed in small steps.
- The program is kept in a compilable and running state throughout the entire process (i.e. after each refactoring step is completed).
- Preservation of behaviour is validated by unit tests (most of the time, just one is used) that must run successfully as soon as each transformation is completed.
- No new functionality is introduced during the refactoring process.

This example is also useful to illustrate how the capabilities of a programming language have a profound influence on the design of programs written in that language, and even on the very idea of what comprises a good design. Though AspectJ is backwards compatible with Java, the two designs are profoundly different. This is compounded by implementation issues. The original Java design uses the Observable and Observer types from Java's java.util API, while the AspectJ design relies on internal collections owned by aspects. Consequently, the structural changes made during the refactoring process are very deep.

This Chapter is structured as follows. Section 7.1 describes the Observer design pattern. Section 7.2 describes the specific instance of Observer that is used as the starting point of the refactoring processes described in this Chapter. Sections 7.3 and 7.4 discuss issues of the Java implementation of that instance. Section 7.5 describes the test used throughout the process to ensure preservation of behaviour. Section 7.6 describes the AspectJ implementation that comprises the desirable end of the refactoring processes. Section 7.7 describes the refactoring processes.

7.1. The Observer Design Pattern

The intent of Observer design pattern (also known as Publish-Subscribe) is to "define a one-to-many dependence between objects so that when one object changes state, all its dependents are notified and updated automatically" [48]. The pattern defines the role of *subject* as the object that generates events that are of interest to several other objects, which play the role of *observers*. Many implementations of this pattern usually include an extra field

[30] This refactoring process is also documented in a eclipse project that includes 33 complete snapshots of the example's code. The project is available in www.di.uminho.pt/~jmf/PUBLI/papers/ObserverExample.zip

in each of the objects playing the subject role, holding the list of its observers. Observers are added to the list by an attach operation and can be removed from the list by a detach operation. When the subject gives rise to one of the interesting events – usually a change in its state – it must call a notify operation, which comprises traversing the list of registered observers and performing the update operation on each of them (see Figure 12, which is similar to that used in [48]).

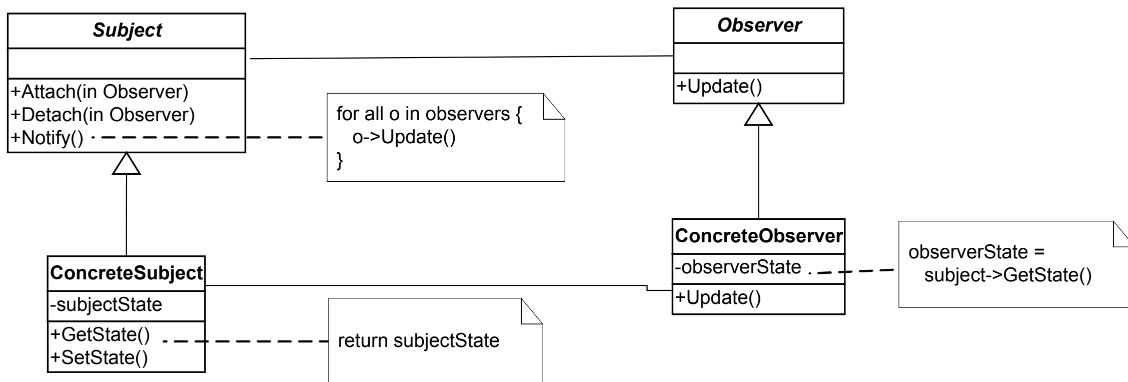

Figure 12: Structure for the Observer Design Pattern.

The example in the GoF book [48] is coded in C++ and therefore is based on abstract classes to represent the roles of subject and observer. With Java, it is usual to represent these roles through interfaces. In each case, the implementing classes are called *concrete subject* and *concrete observer*, respectively.

Each observer class defines the specific reaction of their instances to being notified by the update operation. What qualifies as an interesting event is exclusively determined by the points where the calls to the notify method are made, so programmers must ensure that such calls are made consistently and in all suitable places. In large systems, this may comprise hundreds or thousands of calls, scattered throughout dozens or hundreds of packages. Therefore, ensuring consistency in large systems is hard and error-prone. For the same reason, switching from one implementation to another in large systems is a hard and tedious task.

7.2. The Flower Example

The subject in Eckel's flower example [39] is one instance of a class representing a flower. Its interesting events are its two operations: open its petals and close them. These are observed by instances of two unrelated types: bees and humming birds. When the flower opens its petals, its observers have breakfast. When the flower closes its petals, its observers go to sleep. These reactions are represented by simple messages sent to the console. Each of the two flower operations gives rise to a different observing relationship, as the observers react differently to the two events and it is possible to support one relationship without supporting the other. Reactions of the observers are represented by simple messages sent to the console. The system also ensures that the observers only react once to each operation. For instance, if the flower executes the open operation twice with no close in between, the observers only react to the first open (as we're going to see, this places an additional hurdle to the refactoring process, as the AspectJ implementation did not take this issue into account). The example includes one flower, one bee and one bird.

7.3. The Java Standard API Observer-Observable Protocol

Java's java.util API provides a ready-made implementation of the Observer pattern, comprising the Observer interface and the Observable class.

Subject classes must inherit from Observable, which provides the logic to manage the list of subscribed observers. Subject objects notify their observers of an interesting event by calling the notifyObservers method, with one of the following alternative signatures:

```
public void notifyObservers(Object arg)
public void notifyObservers()          // equivalent to notifyObservers(null)
```

This method only notifies the observers if the object was previously marked as having been changed, by executing the setChanged method. Observer classes must implement the Observer interface, which declares an update method, with the following signature:

```
void update(Observable o, Object arg)
```

Subjects can use the second parameter of type Object to pass data to its observers. In order to be general-purpose, the parameter must accept any type, which in Java means java.lang.Object. This provides the necessary flexibility but has the disadvantage of placing this parameter outside the reach of the type-checker. It is the programmer's responsibility to ensure that subjects and its observers use the same runtime type consistently[31]. The update method contains the actions of the observer objects when they are notified of an interesting event, usually a change in the state of the subject. What qualifies as interesting is exclusively determined by the calls to this method, made by the objects playing the subject role. Programmers using this protocol must ensure that such calls are made in all suitable places. This includes calling the setChanged method. In very large systems, this may mean hundreds or thousands of calls, scattered throughout dozens or hundreds of packages. This of course means that it would be very hard to switch from one implementation to another after the system's structure is in place.

Besides the usual problems of code scattering and tangling, this Java solution also has the following disadvantages:

- Classes playing the Subject role loose the option of inheriting from some other class, because their "single slot of inheritance" is taken by java.util.Observable. The observer participants are less limited because they merely implement the java.util.Observer interface. Even so, this contributes to clutter their 'implements' clause with an interface unrelated to class' primary concern.
- Inheriting from java.util.Observable increases the memory footprint of each instance. Objects playing this role must carry the extra state throughout their entire life cycle, even if they only use it during certain phases.
- Use of inheritance also means that all instances of the subclasses of java.util.Observable will carry the extra state, even if only a subset of the instances participates in observing relationships.
- This mechanism does not separate multiple observing relationships. If the instances of a class play the subject role in various observing relationships, their observers will be notified of the events relating to all of them, and will need to run extra logic in order to distinguish one interesting event from the others. It is possible to ameliorate this problem by making the subject pass itself as argument in the version

[31] This is one of many instances which can benefit from generic types, in order to bring type safety without compromising flexibility.

of notifyObservers with two arguments, but this merely pushes the filtering logic from the subjects to the observers.

7.4. The Original Java Implementation

Listing 24 shows the subject class Flower. Listing 25 shows the observer class Bee and Listing 26 shows the other observer class (Hummingbird), which is similar.

Eckel's design partially circumvents the limitations stated above. The design relies on inner classes to isolate, within each class, the code related to the pattern. Instead of directly extending the Observer or the Observable types, each of the participants includes an inner class that either extends Observable (in the case of subjects) of implements Observer (in the case of the observers). Each participant contains one inner class for each of the observing relationships. Inner classes provide users of Java a very limited form of multiple inheritance, in that inner classes can refer to the members of its enclosing class, even private ones. They are still free to inherit from some other class, so they enable the enclosing class to make a part of itself inherit from a given class while remaining free to use the more traditional use of inheritance. Eckel used this mechanism in both the subject and the observer participants.

This design has the advantage of freeing subjects to inherit from some class useful to their implementations other than java.util.Observable (though this particular example does not take advantage of this). It also avoids cluttering the observer's implements clause with one more interface. The design manages to localise, within each class, the code related to the pattern, but it also produces an even tighter structural relationship between the participants and the roles they play in the pattern. This places additional hurdles in a refactoring process aiming to switch to another design.

What even this clever design cannot achieve is full obliviousness [45] from the pattern roles. All three participant classes betray the *Double Personality* (78) smell, i.e. each participant contains code related to more than one concern. There is still code tangling, because each participant class contains code related to two concerns – the primary concern and the role in the pattern. Any method of the subject class – Flower (see Listing 24) – performing an interesting operation must still include code relative to the subject role. There is still code scattering, because code dealing with the pattern is not modularised. Each participant contains one inner class for each of the observing relationships. There is much duplication[32], which is particularly noticeable in the two observer classes – Bee and Hummingbird (see Listing 25 and Listing 26) – which require four inner classes between them. Each class duplicates the code related to the two observing relationships and each observing relationship requires a duplication of essentially the same logic.

Note that each of the two observing relationships needs to observe both the open and close actions, though for different purposes. This is due to the requirement that an observer only reacts to the first occurrence of an operation. If a flower executes its close method twice without executing open in between, registered observers only react to the first close. Observers of the open operation need to be notified of close, so they know if an open is the first to execute or not. Likewise, observations of close must be notified of open to know when an execution of close is the first or not.

[32] Naturally, we ignore the fact that the two observers are almost identical, as that duplication is specific to the present example.

```
public class Flower {
   private boolean isOpen;
   private OpenNotifier oNotify = new OpenNotifier();
   private CloseNotifier cNotify = new CloseNotifier();

   public Flower() {
      isOpen = false;
   }
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   public void close() { // Closes its petals
      System.out.println("Flower close.");
      isOpen = false;
      cNotify.notifyObservers();
      oNotify.close();
   }
   public Observable opening() {
      return oNotify;
   }
   public Observable closing() {
      return cNotify;
   }
   private class OpenNotifier extends Observable {
      private boolean alreadyOpen = false;
      public void notifyObservers() {
         if(isOpen && !alreadyOpen) {
            setChanged();
            super.notifyObservers();
            alreadyOpen = true;
         }
      }
      public void close() {
         alreadyOpen = false;
      }
   }
   private class CloseNotifier extends Observable {
      private boolean alreadyClosed = false;
      public void notifyObservers() {
         if(!isOpen && !alreadyClosed) {
            setChanged();
            super.notifyObservers();
            alreadyClosed = true;
         }
      }
      public void open() {
         alreadyClosed = false;
      }
   }
}
```

Listing 24: Initial (tangled) form of the Flower subject class.

```
public class Bee {
   private String name;
   private OpenObserver openObsrv = new OpenObserver();
   private CloseObserver closeObsrv = new CloseObserver();

   public Bee(String nm) {
      name = nm;
   }
   // An inner class for observing openings:
   private class OpenObserver implements Observer {
      public void update(Observable ob, Object a) {
         System.out.println(
            "Bee " + name + "'s breakfast time!");
      }
   }
   // Another inner class for closings:
   private class CloseObserver implements Observer{
      public void update(Observable ob, Object a) {
         System.out.println(
            "Bee " + name + "'s bed time!");
      }
   }
   public Observer openObserver() {
      return openObsrv;
   }
   public Observer closeObserver() {
      return closeObsrv;
   }
}
```

Listing 25: Initial (tangled) form of the Bee observer class.

```
public class Hummingbird {
   private String name;
   private OpenObserver openObsrv = new OpenObserver();
   private CloseObserver closeObsrv = new CloseObserver();

   public Hummingbird(String nm) {
      name = nm;
   }
   private class OpenObserver implements Observer{
      public void update(Observable ob, Object a) {
         System.out.println(
            "Hummingbird " + name + "'s breakfast time!");
      }
   }
   private class CloseObserver implements Observer{
      public void update(Observable ob, Object a) {
         System.out.println(
            "Hummingbird " + name + "'s bed time!");
      }
   }
   public Observer openObserver() {
      return openObsrv;
   }
   public Observer closeObserver() {
      return closeObsrv;
   }
}
```

Listing 26: Initial (tangled) version of the Hummingbird observer class

7.5. Unit-Testing the Flower Example

The example uses a JUnit test throughout[33]. It is an adapted version of the original test provided by Eckel. Listing 27 shows a fragment of that test, which in this example also doubles as client code.

```
public class TestObservedFlower extends TestCase {
   Flower f = new Flower();
   Bee ba = new Bee("A"), bb = new Bee("B");
   Hummingbird hx = new Hummingbird("X"), hy = new Hummingbird("Y");

   public void test() {
      f.opening().addObserver(ba.openObserver());
      f.opening().addObserver(bb.openObserver());
      f.opening().addObserver(hx.openObserver());
      f.opening().addObserver(hy.openObserver());

      f.closing().addObserver(ba.closeObserver());
      f.closing().addObserver(bb.closeObserver());
      f.closing().addObserver(hx.closeObserver());
      f.closing().addObserver(hy.closeObserver());
      // Hummingbird Y decides to sleep in:
      f.opening().deleteObserver(
         hy.openObserver());
      // A change that interests observers:
      f.open();
      f.open(); // It's already open, no change.
      // Bee A doesn't want to go to bed:
      f.closing().deleteObserver(
         ba.closeObserver());
      f.close();
      f.close(); // It's already closed; no change
      f.opening().deleteObservers();
      f.open();
      f.close();
   }
```

Listing 27: Test method used throughout as client code.

There was no need for the test to be in plain Java for the purposes of this refactoring example. We took the advantage of AspectJ's capabilities to solve a few hurdles related to the adaptation of the original test. In its original form, Eckel's unit test printed the various messages to the console but did not use JUnit's assertion mechanism. This went against our aim of making the test fully automatic (i.e. without any output to the console), at least when running it in the test mode (it would run in the "normal", or application, mode when executing the static main method). Using plain Java, this would entail removing the test's output to the console, but those messages were also the ones that would need to collect in the test string, so that in the end it could be compared with a standard string. On the other hand, it would be convenient to be able to see the output in the console when running the test in application mode (i.e. calling the test class's static main method).

We solved these hurdles by placing two inner aspects within the test class. The first aspect suppresses output to the console when the test is running in test mode (in order to make it automatic), and leaves the messages in place when it is running in application mode (i.e. when it runs from the context of the static main method). The second collects the output into a string and provides a getter method through which the string can be obtained. It

[33] This test implements an use case, rather than test an individual class or method. For this reason, it is adequate to regard it as a functional test rather than an unit test, even if it is implemented with JUnit. We considered the test sufficient to guarantee preservation of behaviour in this simple example.

looked sensible to place the aspects inside the test class because they are intended to be active only when the test class itself is included in the system's build[34].

The test uses the technique of collecting into a string the test's output, and then comparing it with a standard string containing the expected output. Any changes in the string betray a change in the behaviour of the participant objects. The test was indeed useful to warn of changes in the program's behaviour on several occasions[35].

Though the aspects were originally made with a specific test in mind, it did not escape us that they have a potential to be reusable. For that reason, we placed the generally applicable parts in a separate package (called util). Though we did not specifically strive for fully reusable aspects when they were created, they proved to be sufficiently general to be reused in various refactoring sessions performed on different source bases, including the ones presented in this Chapter. Here, we show them exactly as they were used throughout the refactoring sessions.

The structure of these aspects is the usual in these cases: an abstract aspect containing the reusable parts and concrete aspects containing the case-specific parts. Listing 28 shows the abstract aspect for suppressing output and Listing 29 shows the abstract aspect for collecting output.

```
package util;

/**
 * This aspect supresses output from objects when they do NOT originate
 * from joinpoints captured by the abstract pointcut scopeException()
 */
public abstract aspect OutputSupress {
   /**
    * Specifies a set of joinpoits that are exempt from the
    * output supression.
    */
   protected abstract pointcut scopeException();

   /**
    * Captures calls to console
    */
   public pointcut allSysOutPrintsWithException():
      Pointcuts.allCalls2SystemOutPrints()
      && !scopeException();

   /**
    * Deactivate messages to the console
    */
   void around(): allSysOutPrintsWithException() {
      //null op
   }
}
```

Listing 28: OutputSupress abstract aspect

We dot not consider these aspects fully reusable because not all possible output cases are covered by the pointcuts. Nevertheless, some of the pointcuts used – call2SystemOutPrintln and allCalls2SystemOutPrints, related to calls to System.out –

[34] Note that the fact that an aspect is enclosed within a class does not place limits on the scope of joinpoints it can affect.

[35] The most notable situation in which the unit test warned that something was amiss was when an obvious error was purposely introduced in the code and yet the test passed. It turned out that we were changing the sources from one of the snapshots we were keeping to register the various stages of the refactoring process. This warning was useful to make us be on our guard in relation to this kind of mistake.

showed such a potential to be reused that we felt it worthwhile to place them in a separate aspect containing useful pointcuts, placed in the same package (Listing 30). The intent of this aspect is to hold all pointcuts that promise to be useful in various situations.

The initial form of the complete unit test, including the inner concrete aspects that extend the two abstract aspects OutputSupress and OutputCapture, is shown in Listing 31. The inner aspects define the case-specific pointcuts scopeException and printScope. Note that SpecificOutputCapture declares precedence over SpecificOutputSuppress, ensuring that the first aspect has the opportunity to capture the messages sent to the console before the SpecificOutputSuppress eliminates these joinpoints.

```
package util;

public abstract aspect OutputCapture {
   protected static String resultStr = "";

   public abstract pointcut printScope();

   public pointcut messagesFromSystemOutPrintln(Object message):
      Pointcuts.call2SystemOutPrintln()
      && printScope()
      && args(message);

   /**
    * Capture all messages from System.out.println(Object)
    */
   void around(Object message): messagesFromSystemOutPrintln(message) {
      resultStr += message.toString() + "\n";
      proceed(message);
   }
   public static String getResult() {
      return resultStr;
   }
   public static void clearResult() {
      resultStr = "";
   }
   public static void printResult() {
      System.out.println("\nCAPTURED OUTPUT:[" + resultStr + "]");
   }
}
```

Listing 29: OutputCapture abstract aspect

```
package util;

public aspect Pointcuts {
   public pointcut call2SystemOutPrintln():
      call(public void java.io.PrintStream.println(Object+));

   public pointcut allCalls2SystemOutPrints():
      call(public void java.io.PrintStream.print*(..));
}
```

Listing 30: Pointcut aspect of generally applicable pointcuts

```
public class TestObservedFlower extends TestCase {
   Flower f = new Flower();
   Bee ba = new Bee("A"), bb = new Bee("B");
   Hummingbird hx = new Hummingbird("X"), hy = new Hummingbird("Y");
   public void test() {
      f.opening().addObserver(ba.openObserver());
      f.opening().addObserver(bb.openObserver());
      f.opening().addObserver(hx.openObserver());
      f.opening().addObserver(hy.openObserver());

      f.closing().addObserver(ba.closeObserver());
      f.closing().addObserver(bb.closeObserver());
      f.closing().addObserver(hx.closeObserver());
      f.closing().addObserver(hy.closeObserver());
      // Hummingbird Y decides to sleep in:
      f.opening().deleteObserver(hy.openObserver());
      // A change that interests observers:
      f.open();
      f.open(); // It's already open, no change.
      // Bee A doesn't want to go to bed:
      f.closing().deleteObserver(ba.closeObserver());
      f.close();
      f.close(); // It's already closed; no change
      f.opening().deleteObservers();
      f.open();
      f.close();
      String testStr =
         "Flower open.\n" +
         "Hummingbird X's breakfast time!\n" +
         "Bee B's breakfast time!\n" +
         "Bee A's breakfast time!\n" +
         "Flower open.\n" +
         "Flower close.\n" +
         "Hummingbird Y's bed time!\n" +
         "Hummingbird X's bed time!\n" +
         "Bee B's bed time!\n" +
         "Flower close.\n" +
         "Flower open.\n" +
         "Flower close.\n" +
         "Hummingbird Y's bed time!\n" +
         "Hummingbird X's bed time!\n" +
         "Bee B's bed time!\n";
      Assert.assertEquals(testStr, SpecificOutputCapture.getResult());
   }

   /** This inner aspect supresses output from objects within this package
    *  when they do NOT originate from within the control flow of main()    */
   private static aspect SpecificOutputSuppress extends OutputSupress {
      protected pointcut scopeException():
         cflow(execution(
            public static void TestObservedFlower*.main(String[])
            ));
   }

   private static aspect SpecificOutputCapture extends OutputCapture {
      declare precedence: TestObservedFlower.SpecificOutputCapture,
                          TestObservedFlower.SpecificOutputSuppress;
      public pointcut printScope():
         !within(TestObservedFlower.SpecificOutputCapture);
   }

   public static void main(String args[]) {
      junit.textui.TestRunner.run(TestObservedFlower.class);
   }
}
```

Listing 31: Test class for the Flower example of the Observer pattern

7.6. The Reusable AOP Implementation of Observer

The AspectJ implementation proposed in [60] follows the same structural pattern as the aspects used in the test presented above: an abstract (and potentially reusable) aspect – ObserverProtocol – dealing with the parts common to all cases (see Listing 32), and a concrete subaspect dealing with case-specific parts (see Listing 33 for one specific example[36], taken from [60]). The common parts are:

- The subject and observer roles, modelled by inner (marker) interfaces Subject and Observer.
- Maintenance of a mapping from subjects to observers, implemented with a weak hash map, the perSubjectObservers field.
- The update logic, in which changes in the subject trigger updates in the observers. This is modelled by a single abstract pointcut subjectChange and the advice acting on its joinpoints.

The parts specific to each individual case are:

- Binding the subject and observer roles to the concrete classes, implemented with 'declare parents' clauses.
- The set of changes on the subject that are of interest to its observers, implemented by a concrete definition of the abstract pointcut subjectChange.
- The specific means of updating the observers when the update logic requires it. This is implemented by the updateObserver method.

The most notable thing of the AspectJ implementation is that participant classes become completely oblivious to the roles they play in the pattern. None of the disadvantages mentioned in relation to the Observer-Observable protocol from the Java standard java.lang API applies in this implementation. The classes of participants remain free to inherit from other classes, and instances do not expend any additional memory space when not participating in observing relationships. The mapping between a subject and its observers is maintained by the aspect itself rather than using the mechanism of inter-type declarations, which would affect all instances of the introduced classes, throughout their entire life cycles. The structure managing the mappings of subjects their list of registered observers is defined in the abstract superaspect. One disadvantage of this implementation is a possible[37] degradation in performance in the case of very large systems involving a very large number of participant objects.

[36] The sources referred in [60] – labeled version 1.0 – were later updated with a version 1.1. This later version was used in this example.

[37] No performance measurement is provided in [60].

```
import java.util.WeakHashMap;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

public abstract aspect ObserverProtocol {
   protected interface Subject  { }
   protected interface Observer { }

   private WeakHashMap perSubjectObservers;

   protected List getObservers(Subject s) {
      if (perSubjectObservers == null) {
         perSubjectObservers = new WeakHashMap();
      }
      List observers = (List)perSubjectObservers.get(s);
      if ( observers == null ) {
         observers = new LinkedList();
         perSubjectObservers.put(s, observers);
      }
      return observers;
   }
   public void addObserver(Subject s, Observer o) {
      getObservers(s).add(o);
   }
   public void removeObserver(Subject s, Observer o) {
      getObservers(s).remove(o);
   }

   protected abstract pointcut subjectChange(Subject s);

   after(Subject s): subjectChange(s) {
      Iterator iter = getObservers(s).iterator();
      while ( iter.hasNext() ) {
         updateObserver(s, ((Observer)iter.next()));
      }
   }

   protected abstract void updateObserver(Subject s, Observer o);
}
```

Listing 32: The reusable ObserverProtocol asbtract aspect

```
import java.awt.Color;
import library.ObserverProtocol;

public aspect ColorObserver extends ObserverProtocol {
   declare parents: Point  implements Subject;
   declare parents: Screen implements Observer;

   protected pointcut subjectChange(Subject s):
      call(void Point.setColor(Color)) && target(s);

   protected void updateObserver(Subject s, Observer o) {
      ((Screen)o).display("Screen updated because color changed.");
   }
}
```

Listing 33: One concrete subaspect of ObserverProtocol

7.7. Refactoring Sessions

Note that the refactorings described in the next sections follow only a few of many possible paths. Though the result should always be similar, it is possible to reach it through multiple paths, since each step marks a point from which it is possible to take alternative decisions.

The process presented next has three phases: (1) the extraction of two observing relationships into their own aspects – using *Extract Feature into Aspect* (90) as the general framework, (2) next improve their internal structures – by applying *Tidy Up Internal Aspect Structure* (128), and (3) eliminate various duplications between the two resulting aspects by applying *Extract Superaspect* (129).

The two alternative paths diverge at the start of the second phase. In the first path, we tidy up the aspects and use *Extract Superaspect* (129) to obtain the common superaspect. In the second path, we take advantage of the framework provided by the ObserverProtocol abstract aspect. In this case, there is no need to specifically apply *Extract Superaspect* (129) to the two aspects, because we add the ObserverProtocol aspect early in the course of this refactoring path.

Code fragments are used to illustrate various details. Changes from the previous state are highlighted in bold.

7.7.1 First Phase: Feature Extraction

We start by extracting the relationship related with watching Flower.open. There are three inner classes related with this concern, Flower.OpenNotifier, Bee.OpenObserver and Hummingbird.OpenObserver (see Listings 23-25). We apply *Extract Inner Class into Standalone* (99) to Flower.OpenNotifier, yielding the following standalone class:

```
import java.util.Observable;
import java.util.Observer;

public class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean alreadyOpen = false;
   public OpenNotifier(Flower flower) {
      _enclosing = flower;
   }
   public void notifyObservers() {
      if(_enclosing.isOpen() && !this.alreadyOpen) {
         this.setChanged();
         super.notifyObservers();
         this.alreadyOpen = true;
      }
   }
   public void close() {
      this.alreadyOpen = false;
   }
}
```

This refactoring also entailed the prior extraction, using *Extract Method* ([47], p.110), of a getter method for Flower.isOpen:

```
public class Flower {
   private boolean isOpen;
   private OpenNotifier oNotify = new OpenNotifier(this);
   //...
   boolean isOpen() {
      return isOpen;
   }
```

Next, we would like to do the same with Bee.OpenObserver and Hummingbird.OpenObserver but there are two problems here. One is that each contains an action – print a message to the console – that should be considered as part of the enclosing class's primary functionality. This is dealt with by applying *Extract Method* ([47], p.110) to the code fragment in each class. On Bee is as follows:

```
public class Bee {
   //...
   void breakfastTime() {
      System.out.println("Bee " + name + "'s breakfast time!");
   }
   // An inner class for observing openings:
   private class OpenObserver implements Observer {
      public void update(Observable ob, Object a) {
         breakfastTime();
      }
   }
```

On Hummingbird the refactoring is similar:

```
public class Hummingbird {
   //...
   void breakfastTime() {
      System.out.println("Hummingbird " + name + "'s breakfast time!");
   }
   private class OpenObserver implements Observer{
      public void update(Observable ob, Object a) {
         breakfastTime();
      }
   }
```

The other problem is that both classes would have the same name after being turned into standalones. Since they are almost identical, it would be simpler to make them the same. However, each must hold a field referring to its enclosing class, which are of different types. Our solution is to use *Extract Interface* ([47], p.341) so that the field of inner class can be of the resulting interface type.

```
public interface BreakfastTaker {
   public void breakfastTime();
}
```

This in turn forces us to make the breakfastTime methods public.

```
public class Bee implements BreakfastTaker {
   //...
   public void breakfastTime() {
      //...

public class Hummingbird implements BreakfastTaker {
   //...
   public void breakfastTime() {
      //...
```

Next, we apply *Extract Inner Class into Standalone* (99) and use the new interface as the type of the enclosing object.

```
import java.util.Observable;
import java.util.Observer;

public class OpenObserver implements Observer {
   private BreakfastTaker _enclosing;
   public OpenObserver(BreakfastTaker enclosing) {
      _enclosing = enclosing;
   }
   public void update(Observable ob, Object a) {
      _enclosing.breakfastTime();
   }
}

public class Bee implements BreakfastTaker {
   //...
   private OpenObserver openObsrv = new OpenObserver(this);

public class Hummingbird implements BreakfastTaker {
   //...
   private OpenObserver openObsrv = new OpenObserver(this);
```

The code if now ripe for the extraction of the various elements to an aspect. The blank aspect ObservingOpen is created.

```
public aspect ObservingOpen {
}
```

Next, we apply the following refactorings:

- *Move Field from Class to Inter-type* (104) to field Flower.oNotify. The private access of oNotify is temporarily relaxed to package-protected.
- *Move Method from Class to Inter-type* (106) to the Flower.opening method.
- *Extract Fragment into Advice* (94) to the call to Flower.oNotify.notifyObservers.
- *Extract Fragment into Advice* (94) to the call to Flower.oNotify.close.

These refactorings move all code using oNotify to the aspect, so it is now possible to make it private again. The aspect now has the following contents:

```
import java.util.Observable;

public aspect ObservingOpen {
   private OpenNotifier Flower.oNotify = new OpenNotifier(this);
   public Observable Flower.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning : flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   pointcut flowerClose(Flower flower):
      execution(void close()) && this(flower);
   after(Flower flower): flowerClose(flower) {
      flower.oNotify.close();
   }
}
```

Flower became clean of any code related to the first observing relationship:

```
import java.util.Observable;
import java.util.Observer;

public class Flower {
   private boolean isOpen;
   private CloseNotifier cNotify = new CloseNotifier();

   public Flower() {
      isOpen = false;
   }
   boolean isOpen() {
      return isOpen;
   }
   private void setIsOpen(boolean newValue) {
      isOpen = newValue;
   }
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      setIsOpen(true);
      cNotify.open();
   }
   public void close() { // Closes its petals
      System.out.println("Flower close.");
      setIsOpen(false);
      cNotify.notifyObservers();
   }
   public Observable closing() {
      return cNotify;
   }
   private class CloseNotifier extends Observable {
      private boolean alreadyClosed = false;
      public void notifyObservers() {
         if(!isOpen() && !alreadyClosed) {
            setChanged();
            super.notifyObservers();
            alreadyClosed = true;
         }
      }
      public void open() {
         alreadyClosed = false;
      }
   }
}
```

The next step is to extract from the observer classes Bee and Hummingbird all the remaining elements related to this concern. We apply *Move Field from Class to Inter-type* (104) to Bee.openObsrv. As often happens with private fields, this forces us to relax the access from private to package-protected. As recommended by *Move Field from Class to Inter-type* (104), the following 'declare warning' was created:

```
   declare warning:
      get(OpenObserver Bee.openObsrv)
      && !within(ObservingOpen):
      "field Bee.openObsrv accessed outside aspect.";
```

The 'declare warning' signalled an occurrence of the field outside the aspect, in the Bee.openObserver method. This method also belongs to this concern, so we move it next, using *Move Method from Class to Inter-type* (106). After compiling again, there was no more warnings, so the 'declare warning' is removed and the access to openObsrv field is made private again. Next, similar refactorings are applied to Hummingbird. After this, the observers are devoid of any code related the first observation relationship, save for the implements clause referring to BreakfastTaker:

```
public class Bee implements BreakfastTaker {
   private String name;
   private CloseObserver closeObsrv = new CloseObserver();

   public Bee(String nm) {
      name = nm;
   }
   public void breakfastTime() {
      System.out.println("Bee " + name + "'s breakfast time!");
   }
   // Another inner class for closings:
   private class CloseObserver implements Observer{
      public void update(Observable ob, Object a) {
         System.out.println("Bee " + name + "'s bed time!");
      }
   }
   public Observer closeObserver() {
      return closeObsrv;
   }
}

public class Hummingbird implements BreakfastTaker {
   private String name;
   private CloseObserver closeObsrv = new CloseObserver();

   public Hummingbird(String nm) {
      name = nm;
   }
   public void breakfastTime() {
      System.out.println("Hummingbird " + name + "'s breakfast time!");
   }
   private class CloseObserver implements Observer{
      public void update(Observable ob, Object a) {
         System.out.println("Hummingbird " + name + "'s bed time!");
      }
   }
   public Observer closeObserver() {
      return closeObsrv;
   }
}
```

At this point, the aspect contains the following code:

```
import java.util.Observable;
import java.util.Observer;

public aspect ObservingOpen {
   private OpenNotifier Flower.oNotify = new OpenNotifier(this);
   private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
   private OpenObserver Bee.openObsrv = new OpenObserver(this);
   public Observable Flower.opening() {
      return oNotify;
   }
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Flower flower): execution(void open()) && this(flower);
   after(Flower flower) returning : flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   pointcut flowerClose(Flower flower):
      execution(void close()) && this(flower);
   after(Flower flower): flowerClose(flower) {
      flower.oNotify.close();
   }
   public Observer Bee.openObserver() {
      return openObsrv;
   }
}
```

The next task comprises the extraction of the second observing relationship into a second aspect, which comprises a similar sequence of steps. Naturally, this exposes a significant amount of duplication between the two aspects. This can be dealt with after both concerns are modularised within their respective aspects. The following steps are:

- Apply *Extract Inner Class to Standalone* (99) to the CloseNotifier class within Flower.
- Create a new blank aspect ObservingClose.
- Apply *Move Field from Class to Inter-type* (104) to the field Flower.cNotify, whose access is temporarily relaxed from private to package-protected. Using this refactoring entails creating a 'declare warning' which exposes three points in Flower still using the field.
- Apply *Move Method From Class to Inter-type* (106) to Flower.closing. This removes one of the warnings. The import statements in Flower can now be removed.
- Apply *Extract Fragment into Advice* (94) to the calls to cNotify.open and cNotify. notifyObservers. This removes the two remaining warnings, so the field Flower.cNotify is made private again and the 'declare warning' is removed.

From this point on, Flower is clean of any code related to observing relationships (see Listing 34).

```
public class Flower {
   private boolean isOpen;

   public Flower() {
      isOpen = false;
   }
   boolean isOpen() {
      return isOpen;
   }
   private void setIsOpen(boolean newValue) {
      isOpen = newValue;
   }
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      setIsOpen(true);
   }
   public void close() { // Closes its petals
      System.out.println("Flower close.");
      setIsOpen(false);
   }
}
```

Listing 34: Clean version of the Flower class.

Next, we deal with the remaining code in the observer participants, Bee and Hummingbird. The first thing is to unify both CloseObserver inner classes within Bee and Hummingbird so that *Extract Inner Class into Standalone* (99) can be applied to both classes simultaneously, yielding a single standalone class. This entails (1) applying *Extract Method* ([47], p.110) to create the bedtimeSleep method in each of them, (2) use *Extract Interface* ([47], p.341) to extract BedtimeSleep. This repeats the actions that resulted in the extraction of the breakfastTime method and in the creation of the BreakfastTaker interface:

```
public interface BedtimeSleeper {
   public void bedtimeSleep();
}
```

Now we can use *Extract Inner Class into Standalone* (99) to both CloseObserver inner classes to produce the following common standalone class:

```
import java.util.Observer;
import java.util.Observable;

public class CloseObserver implements Observer {
   private BedtimeSleeper _enclosing;
   public CloseObserver(BedtimeSleeper enclosing) {
      _enclosing = enclosing;
   }
   public void update(Observable ob, Object a) {
      _enclosing.bedtimeSleep();
   }
}
```

We then move all remaining members related to the extracted concern to the second aspect:

- Apply *Move Field From Class to Inter-type* (104) to Bee.closeObsrv.
- Apply *Move Method From Class to Inter-type* (106) to Bee.closeObserver.
- Apply *Move Field From Class to Inter-type* (104) to Hummingbird.closeObsrv.
- Apply *Move Method From Class to Inter-type* (106) to Hummingbird.closeObserver.

The import statements in Bee and Hummingbird can now be removed. The only remaining code in the participants relating to the observing relationships is the implements clauses referring to BreakfastTaker and BedtimeSleeper. We now use *Replace Implements with Declare Parents* (108) to both Bee and Hummingbird, relative to these interfaces.

```
public aspect ObservingOpen {
   declare parents: (Bee || Hummingbird) implements BreakfastTaker;

public aspect ObservingClose {
   declare parents: (Bee || Hummingbird) implements BedtimeSleeper;
```

Now all participants are completely free of any code related to extracted concerns (see Listings 34-36).

```
public class Bee {
   private String name;

   public Bee(String nm) {
      name = nm;
   }
   public void breakfastTime() {
      System.out.println("Bee " + name + "'s breakfast time!");
   }
   public void bedtimeSleep() {
      System.out.println( "Bee " + name + "'s bed time!");
   }
}
```

Listing 35: Clean version of the Bee class

```
public class Hummingbird {
   private String name;

   public Hummingbird(String nm) {
      name = nm;
   }
   public void breakfastTime() {
      System.out.println("Hummingbird " + name + "'s breakfast time!");
   }
   public void bedtimeSleep() {
      System.out.println("Hummingbird " + name + "'s bed time!");
   }
}
```

Listing 36: Clean version of the Hummingbird class

The refactorings made until now cleaned the participant's code but it also created several standalone classes and interfaces that provide little functionality. In addition, these are used only by the aspects. Therefore we proceed to inline them, which has the advantage that absolutely all the code related to the concerns is now within the two aspects, which makes it easier to reason with.

We thought of inlining the interfaces first, but then we noticed we couldn't, because the former inner classes OpenObserver and CloseObserver depend on them. So we inline these classes first, as well as OpenNotifier and CloseNotifier, using *Inline Class within Aspect* (102) for each of them in turn. Next, we inline the BreakfastTaker and BedtimeSleeper interfaces, using *Inline Interface within Aspect* (103). All code related to the two concerns is finally completely modularised within their respective aspects (see Listing 37).

```
import java.util.Observable;
import java.util.Observer;

public aspect ObservingOpen {
   private interface BreakfastTaker {
      public void breakfastTime();
   }
   declare parents: (Bee || Hummingbird) implements BreakfastTaker;

   static class OpenNotifier extends Observable {
      private Flower _enclosing;
      private boolean alreadyOpen = false;
      public OpenNotifier(Flower flower) {
         _enclosing = flower;
      }
      public void notifyObservers() {
         if(_enclosing.isOpen() && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
      public void close() {
         this.alreadyOpen = false;
      }
   }
   static class OpenObserver implements Observer {
      private BreakfastTaker _enclosing;
      public OpenObserver(BreakfastTaker enclosing) {
         _enclosing = enclosing;
      }
      public void update(Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
   private OpenNotifier Flower.oNotify = new OpenNotifier(this);
   private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
   private OpenObserver Bee.openObsrv = new OpenObserver(this);

   public Observable Flower.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning : flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   pointcut flowerClose(Flower flower):
      execution(void close()) && this(flower);
   after(Flower flower): flowerClose(flower) {
      flower.oNotify.close();
   }

   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public java.util.Observer Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

Listing 37: ObservingOpen just after the extraction process.

7.7.2 Second Phase: Restructure the Internals of the Aspect

As can be attested from examining Listing 37, the internal structure of the aspects is confusing. It contains much duplication, as well as several inner classes and interfaces for whose existence there is no longer a compelling reason (particularly if we want to do without the Observer/ Observable pair from java.util). In addition, both aspects betray the *Aspect Laziness* (80) smell. The next phase of this refactoring process is to improve the internal structure of the aspects, by eliminating duplication and *Aspect Laziness* (80).

Let's briefly consider the available options when working with a traditional OO system. Suppose we're dealing with a concern whose implementation code is scattered throughout many classes and packages and we would like to replace it. What would be the right approach? An adequate answer, particularly when the system is large, would be to add a new layer abstracting the details of the existing implementation. This would make the scattered implementation easier to replace, but it would entail the patient refactoring of the system until the new layer completely hides all specific details. The refactoring process would be supported by tests targeting the new layer. When the layer is in place, the developers could develop a new implementation, side-by-side with the old one, against the new layer's interface. They would leverage the tests so that they could run them against both the old implementation and the new. As soon as the new implementation is complete, it becomes possible to switch modules and rebuild the system with the new implementation. With large systems, such a process can take months.

When the concern is modularised, we have more options. We can (1) continue to refactor the aspects' internals in order to transform the existing implementation into the one we want, but it is also feasible to (2) simply develop a new aspect from scratch or take an off-the-shelf solution, and plug it in the system in place of the old. What is shown next is the first option.

Tidying Up the Aspect

The next step is to use *Tidy Up Internal Aspect Structure* (128) on each of them in turn. Not only will this make their internal structures better organised, it will make them more amenable to later apply *Extract Superaspect* (129), in order to eliminate the duplication that exists between them. We next show the refactoring of one aspect[38], ObservingOpen. When that process is completed, a similar one is carried out on ObservingClose.

We start by using *Generalise Target Type with Marker Interface* (113) to expose the duplication in inter-type declarations resulting from *Extract Feature into Aspect* (90). This entails creating the inner marker interfaces within the aspects that represent the pattern roles – subject and observer.

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;
```

This causes a name conflict because the aspect now contains two elements named Observer. We resolve this by removing the import to java.util.Observer and making all references specify the full compound name.

[38] We adopted this strategy to ensure that if some hurdle would arise, there would be a need to backtrack only once. Provided the first refactoring is successful, it is easier to repeat the steps with the second case, relying on the experience gained from the first.

```
import java.util.Observable;
import java.util.Observer;

public aspect ObservingOpen {
   //...
   static class OpenNotifier extends Observable {
      //...
   }
   static class OpenObserver implements Observer {
      //...
   }
   //...
   public Observable Flower.opening() {
      return oNotify;
   }
   //...
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

```
import java.util.Observable;
import java.util.Observer;

public aspect ObservingOpen {
   //...
   static class OpenNotifier extends java.util.Observable {
      //...
   }
   static class OpenObserver implements java.util.Observer {
      //...
   }
   //...
   public java.util.Observable Flower.opening() {
      return oNotify;
   }
   //...
   public java.util.Observer Bee.openObserver() {
      return openObsrv;
   }
   public java.util.Observer Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

We first apply *Generalise Target Type with Marker Interface* (113) to the Flower type. We replace all references to Flower with Subject, including inside the inner class OpenNotifier. This results in a compiler error, because the Subject interface does not declare the isOpen method, used by OpenNotifier.

```
public aspect ObservingOpen {
   //...
   static class OpenNotifier extends java.util.Observable {
      private Subject _enclosing;
      private boolean alreadyOpen = false;

      public void notifyObservers() {
         if(_enclosing.isOpen() && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
```

We use *Extend Marker Interface with Signature* (113) as a stopgap (since we plan to remove OpenNotifier and the other inner classes shortly after).

```
public aspect ObservingOpen {
   //...
   public abstract boolean Subject.isOpen();
```

This in turn forces us to make the Flower.isOpen method public.

```
public class Flower {
   //...
   public boolean isOpen() {
      return isOpen;
   }
```

We next apply *Generalise Target Type with Marker Interface* (113) to Bee and Hummingbird. This enables us to remove the inner interface BreakfastTaker, because we can now use Observer in its place. We also need to use *Extend Marker Interface with Signature* (113) again, to extend Observer with the case-specific signature of breakfastTime.

```
   private interface BreakfastTaker {
      public void breakfastTime();
   }

   declare parents: (Bee || Hummingbird) implements BreakfastTaker;

   static class OpenObserver implements java.util.Observer {
      private BreakfastTaker _enclosing;
      public OpenObserver(BreakfastTaker enclosing) {
         _enclosing = enclosing;
      }
      public void update(java.util.Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
```

```
   private interface BreakfastTaker {
      public void breakfastTime();
   }
   public abstract void Observer.breakfastTime();

   static class OpenObserver implements java.util.Observer {
      private Observer _enclosing;
      public OpenObserver(Observer enclosing) {
         _enclosing = enclosing;
      }
      public void update(java.util.Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
```

This step also eliminated some duplication in the openObserver method, which is introduced twice (to the Bee and Hummingbird classes).

```
   public java.util.Observer Bee.openObserver() {
      return openObsrv;
   }
   public java.util.Observer Hummingbird.openObserver() {
      return openObsrv;
   }
```

```
   public java.util.Observer Observer.openObserver() {
      return openObsrv;
   }
```

The aspect now only refers to the concrete participants in the 'declare parents' clause (see Listing 38).

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;

   public abstract boolean Subject.isOpen();

   public abstract void Observer.breakfastTime();

   static class OpenNotifier extends java.util.Observable {
      private Subject _enclosing;
      private boolean alreadyOpen = false;
      public OpenNotifier(Subject flower) {
         _enclosing = flower;
      }
      public void notifyObservers() {
         if(_enclosing.isOpen() && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
      public void close() {
         this.alreadyOpen = false;
      }
   }
   static class OpenObserver implements java.util.Observer {
      private Observer _enclosing;
      public OpenObserver(Observer enclosing) {
         _enclosing = enclosing;
      }
      public void update(java.util.Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
   private OpenNotifier Subject.oNotify = new OpenNotifier(this);
   private OpenObserver Observer.openObsrv = new OpenObserver(this);

   public java.util.Observable Subject.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Subject subject):
      execution(void open()) && this(subject);
   after(Subject subject) returning : flowerOpen(subject) {
      subject.oNotify.notifyObservers();
   }
   pointcut flowerClose(Subject subject):
      execution(void close()) && this(subject);
   after(Subject subject): flowerClose(subject) {
      subject.oNotify.close();
   }

   public java.util.Observer Observer.openObserver() {
      return openObsrv;
   }
}
```

Listing 38: ObservingOpen aspect after using *Generalise Target Type with Marker Interface*

The next step is to remove *Aspect Laziness* (80), by replacing the inter-type state and behaviour with similar, but dynamic functionality. When everything is in place, we can replace in the client code (in this case the JUnit test) the calls to the original inter-type methods to calls to new aspect methods. To some extent, what follows is an elaborated instance of *Replace Inter-type Field with Aspect Map* (118), using *Replace Inter-type Method with Aspect Method* (125) as a follow-up. The difference in this case is that they target inner classes rather than inter-type fields. This step adds a mapping structure to the aspect, as well as its associated logic.

```
import java.util.WeakHashMap;
import java.util.List;
import java.util.ArrayList;

public aspect ObservingOpen {
   //...
   private WeakHashMap subject2ObserversMap = new WeakHashMap();

   private List getObservers(Subject subject) {
      List observers = (List)subject2ObserversMap.get(subject);
      if(observers == null) {
         observers = new ArrayList();
         subject2ObserversMap.put(subject, observers);
      }
      return observers;
   }
   public void addObserver(Subject subject, Observer observer) {
      List observers = getObservers(subject);
      if(!observers.contains(observer))
         observers.add(observer);
      subject2ObserversMap.put(subject, observers);
   }
   public void removeObserver(Subject subject, Observer observer) {
      getObservers(subject).remove(observer);
   }
   public void clearObservers(Subject subject) {
      getObservers(subject).clear();
   }
```

As prescribed in *Replace Inter-type Method with Aspect Method* (125), we add a notifyObservers method to the aspect, providing the same functionality as OpenNotifier.notifyObservers. The new method uses a new boolean field that we introduce to Subject, used for the same purposes as OpenNotifier.

```
   private boolean Subject.alreadyOpen = false;

   private void notifyObservers(Subject subject) {
      if(subject.isOpen() && !subject.alreadyOpen) {
         subject.alreadyOpen = true;
         List observers = getObservers(subject);
         for(ListIterator it = observers.listIterator(); it.hasNext();) {
            ((Observer)it.next()).breakfastTime();
         }
      }
   }
```

Also as prescribed in *Replace Inter-type Method with Aspect Method* (125), we add a 'declare warning' exposing all places where the old logic is placed. This 'declare warning' targets the Subject.opening method, which is the accessor method for the instance of the inner class OpenNotifier.

```
   declare warning: call(java.util.Observable opening()):
      "opening() called here.";
```

Compiling again exposes six warnings, all placed in the unit test. We proceed to replace the original calls with calls to the aspect logic:

```
public void test() {
   f.opening().addObserver(ba.openObserver());
   f.opening().addObserver(bb.openObserver());
   f.opening().addObserver(hx.openObserver());
   f.opening().addObserver(hy.openObserver());

   f.closing().addObserver(ba.closeObserver());
   f.closing().addObserver(bb.closeObserver());
   f.closing().addObserver(hx.closeObserver());
   f.closing().addObserver(hy.closeObserver());
   // Hummingbird Y decides to sleep in:
   f.opening().deleteObserver(hy.openObserver());

   // A change that interests observers:
   f.open();
   f.open(); // It's already open, no change.
   // Bee A doesn't want to go to bed:
   f.closing().deleteObserver(ba.closeObserver());
   f.close();
   f.close(); // It's already closed; no change
   f.opening().deleteObservers();
```

```
public void test() {
   ObservingOpen.aspectOf().addObserver(f, ba);
   ObservingOpen.aspectOf().addObserver(f, bb);
   ObservingOpen.aspectOf().addObserver(f, hx);
   ObservingOpen.aspectOf().addObserver(f, hy);

   f.closing().addObserver(ba.closeObserver());
   f.closing().addObserver(bb.closeObserver());
   f.closing().addObserver(hx.closeObserver());
   f.closing().addObserver(hy.closeObserver());
   // Hummingbird Y decides to sleep in:
   ObservingOpen.aspectOf().removeObserver(f, hy);

   // A change that interests observers:
   f.open();
   f.open(); // It's already open, no change.
   // Bee A doesn't want to go to bed:
   f.closing().deleteObserver(ba.closeObserver());
   f.close();
   f.close(); // It's already closed; no change
   ObservingOpen.aspectOf().clearObservers(f);
```

When we run the test, the red flag appears: the test failed. After analysing the results, we learn that the problem is due to the two implementations traversing the list of observers in *opposite* orders. This is not an important issue in this case, so we solve the problem by reversing the order in which the observers are subscribed:

```
ObservingOpen.aspectOf().addObserver(f, hy);
ObservingOpen.aspectOf().addObserver(f, hx);
ObservingOpen.aspectOf().addObserver(f, bb);
ObservingOpen.aspectOf().addObserver(f, ba);
```

We compile and test again: the test passes. If we do not attach importance to the order in which the observers are notified, the refactored implementation can now be considered as providing the same behaviour as before. After deleting all the code related to the original implementation, ObservingOpen is as shown in Listing 39.

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;

   public abstract boolean Subject.isOpen();
   public abstract void Observer.breakfastTime();

   static class OpenNotifier extends java.util.Observable {
      private Subject _enclosing;
      private boolean alreadyOpen = false;
      public OpenNotifier(Subject flower) {
         _enclosing = flower;
      }
      public void notifyObservers() {
         if(_enclosing.isOpen() && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
      public void close() {
         this.alreadyOpen = false;
      }
   }
   static class OpenObserver implements java.util.Observer {
      private Observer _enclosing;
      public OpenObserver(Observer enclosing) {
         _enclosing = enclosing;
      }
      public void update(java.util.Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
   private OpenNotifier Subject.oNotify = new OpenNotifier(this);
   private OpenObserver Observer.openObsrv = new OpenObserver(this);

   public java.util.Observable Subject.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Subject subject): execution(void open()) &&
this(subject);
   after(Subject subject) returning : flowerOpen(subject) {
      subject.oNotify.notifyObservers();
   }
   pointcut flowerClose(Subject subject): execution(void close()) &&
this(subject);
   after(Subject subject): flowerClose(subject) {
      subject.oNotify.close();
   }
   public java.util.Observer Observer.openObserver() {
      return openObsrv;
   }
}
```

Listing 39: ObservingOpen after being completely tidied up

Tidying Up the Second Aspect

Improving the internal structure of ObservingClose takes essentially the same steps as with ObservingOpen. Briefly, these are:

- Eliminate the imports to java.util.Observable and java.util.Observer, changing the code to make direct references to the complete name.
- Start applying *Generalise Target Type with Marker Interface* (113), which entails the prior creation of private inner interfaces Observer and Subject.
- Apply *Generalise Target Type with Marker Interface* (113) to Flower: replacing references to this type with Subject. *Extend Marker Interface with Signature* (113) is used to introduce the isOpen method to Subject, as was done with the first aspect.
- Apply *Generalise Target Type with Marker Interface* (113) to Bee and Hummingbird, which are replaced by Observer. During this process the BedtimeSleeper interface is eliminated, as well as the corresponding 'declare parents'. *Extend Marker Interface with Signature* (113) is used again to introduce the bedtimeSleep method to Observer.
- Use *Replace Inter-type Field with Aspect Map* (118) with *Replace Inter-type Method with Aspect Method* (125) to add the new implementation to ObservingClose. Following *Replace Inter-type Method with Aspect Method* (125), a 'declare warning' is added to expose calls to the closing accessor method for the CloseNotifier object:

```
declare warning:
   call(java.util.Observable closing()):
   "closing() called here.";
```

- Following the places exposed by the 'declare warning', the calls in client (test) code are replaced. Again, we reverse the order in which the observers are registered. We remove the 'declare warning' and compile, and the test runs successfully.

After these transformations, ObservingClose is as shown in Listing 40.

```
import java.util.WeakHashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.ListIterator;

public aspect ObservingClose {
   private interface Subject {}
   private interface Observer {}

   public abstract boolean Subject.isOpen();
   public abstract void Observer.bedtimeSleep();
   private boolean Subject.alreadyClosed = false;

   private WeakHashMap subject2ObserversMap = new WeakHashMap();

   private List getObservers(Subject subject) {
      List observers = (List)subject2ObserversMap.get(subject);
      if(observers == null) {
         observers = new ArrayList();
         subject2ObserversMap.put(subject, observers);
      }
      return observers;
   }
   public void addObserver(Subject subject, Observer observer) {
      List observers = getObservers(subject);
      if(!observers.contains(observer))
         observers.add(observer);
      subject2ObserversMap.put(subject, observers);
   }
   public void removeObserver(Subject subject, Observer observer) {
      getObservers(subject).remove(observer);
   }
   public void clearObservers(Subject subject) {
      getObservers(subject).clear();
   }
   private void notifyObservers(Subject subject) {
      if(!subject.isOpen() && !subject.alreadyClosed) {
         subject.alreadyClosed = true;
         List observers = getObservers(subject);
         for(ListIterator it = observers.listIterator(); it.hasNext();) {
            ((Observer)it.next()).bedtimeSleep();
         }
      }
   }
   pointcut flowerOpen(Subject subject):
      execution(void open()) && this(subject);
   after(Subject subject) returning : flowerOpen(subject) {
      subject.alreadyClosed = false;
   }
   pointcut flowerClose(Subject subject):
      execution(void close()) && this(subject);
   after(Subject subject): flowerClose(subject) {
      notifyObservers(subject);
   }

   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;
}
```

Listing 40: ObservingClose after being completely tidied up

7.7.3 Third Phase: Extracting Common Code to a Superaspect

Taken individually, the aspects just refactored are better formed. Together, they betray a strong dose of the *Duplicated Code* smell. As Beck and Fowler state in [47], page 76: "If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them". Therefore, the next task is clear: eliminate the *Duplicated Code* smell, by eliminating the duplication that exists between ObservingOpen and ObservingClose. This is done using *Extract Superaspect* (129) to create a superaspect and pull the common logic up to it. This is done through the following steps:

- Create the ObservingRelationships blank abstract aspect.
- Aspects ObservingOpen and ObservingClose are made to extend ObservingRelationships.
- Use *Pull Up Marker Interface* (132) on the Subject and Observer inner interfaces of both aspects, to move them to ObservingRelationships. Their access is relaxed from private to protected.
- Use *Pull Up Field* ([47], p.320) on the subject2ObserversMap fields of both aspects.
- Use *Pull Up Method* ([47], p.322) on the methods getObservers, addObserver, removeObserver and clearObservers of both aspects.

We would like to use *Pull Up Method* ([47], p.322) on the notifyObservers method as well, but this depends on case-specific members in various points of the method. The notifyObservers method from ObservingOpen uses the isOpen and breakfastTime methods and the alreadyOpen field, and ObservingClose has similar dependencies. For this reason we leave an abstract declaration of notifyObservers in the superaspect.

We also have the option of using *Pull Up Pointcut* (133) to the flowerOpen and flowerClose pointcuts, but as they are case-specific. We should at most place abstract declarations in the superaspect. We refrain from doing that at this stage. This is one of the advantages of refactoring: decisions are not set in stone, and one can always change its mind later, and then refactor. Therefore, the aspects take the final forms as shown in Listing 41, Listing 42 and Listing 43.

```
import java.util.WeakHashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.ListIterator;

public abstract aspect ObservingRelationships {
   protected interface Subject {}
   protected interface Observer {}

   protected WeakHashMap subject2ObserversMap = new WeakHashMap();
   protected List getObservers(Subject subject) {
      List observers = (List)subject2ObserversMap.get(subject);
      if(observers == null) {
         observers = new ArrayList();
         subject2ObserversMap.put(subject, observers);
      }
      return observers;
   }
   public void addObserver(Subject subject, Observer observer) {
      List observers = getObservers(subject);
      if(!observers.contains(observer))
         observers.add(observer);
      subject2ObserversMap.put(subject, observers);
   }
   public void removeObserver(Subject subject, Observer observer) {
      getObservers(subject).remove(observer);
   }
   public void clearObservers(Subject subject) {
      getObservers(subject).clear();
   }
   protected abstract void notifyObservers(Subject subject);
}
```

Listing 41: ObservingRelationships abstract aspect

```
import java.util.WeakHashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.ListIterator;

public aspect ObservingOpen extends ObservingRelationships {
   public abstract boolean Subject.isOpen();
   public abstract void Observer.breakfastTime();
   private boolean Subject.alreadyOpen = false;

   protected void notifyObservers(Subject subject) {
      if(subject.isOpen() && !subject.alreadyOpen) {
         subject.alreadyOpen = true;
         List observers = getObservers(subject);
         for(ListIterator it = observers.listIterator(); it.hasNext();) {
            ((Observer)it.next()).breakfastTime();
         }
      }
   }
   private pointcut flowerOpen(Subject subject):
      execution(void open()) && this(subject);
   after(Subject subject) returning : flowerOpen(subject) {
      notifyObservers(subject);
   }
   protected pointcut flowerClose(Subject subject):
      execution(void close()) && this(subject);
   after(Subject subject): flowerClose(subject) {
      subject.alreadyOpen = false;
   }

   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;
}
```

Listing 42: Final form of the ObservingOpen aspect

```
import java.util.WeakHashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.ListIterator;

public aspect ObservingClose extends ObservingRelationships {
   public abstract boolean Subject.isOpen();
   public abstract void Observer.bedtimeSleep();
   private boolean Subject.alreadyClosed = false;

   protected void notifyObservers(Subject subject) {
      if(!subject.isOpen() && !subject.alreadyClosed) {
         subject.alreadyClosed = true;
         List observers = getObservers(subject);
         for(ListIterator it = observers.listIterator(); it.hasNext();) {
            ((Observer)it.next()).bedtimeSleep();
         }
      }
   }
   private pointcut flowerOpen(Subject subject):
      execution(void open()) && this(subject);
   after(Subject subject) returning : flowerOpen(subject) {
      subject.alreadyClosed = false;
   }
   protected pointcut flowerClose(Subject subject):
      execution(void close()) && this(subject);
   after(Subject subject): flowerClose(subject) {
      notifyObservers(subject);
   }

   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;
}
```

Listing 43: Final form of the ObservingClose aspect

7.7.4 Alternative Refactoring Path

The previous sections show how to use refactoring to gradually transform an implementation in Java into an implementation in AspectJ. However, a reusable aspect providing the same functionality [60] – ObserverProtocol – is already available. It is reasonable to ask whether would it be possible to take advantage of this aspect while refactoring the aspects. That can indeed be accomplished, by including the ObserverProtocol abstract aspect at a certain point in the refactoring process. This comprises an alternative path to the transformations presented in the previous sections.

We start at point at which the extraction process (section 7.7.1) was completed. The existing structure must be tidied up as was done previously, but in this case ObserverProtocol is introduced as a ready-made superaspect, rather than adding similar functionality to the existing aspects, which would later need to be extracted into its own superaspect. This refactoring process involves adding fewer amounts of new code compared to the previous one, because the framework is already set by ObserverProtocol. Only the concrete parts are missing.

Tidying Up the Aspect

We again use the strategy of tackling one aspect first, and then the other. We start by using *Tidy Up Internal Aspect Structure* (128) on ObervingOpen. This refactoring prescribes the use of *Generalise Target Type with Marker Interface* (113), which requires adding the marker interfaces representing the participant roles in the pattern – Subject and Observer. This time we don't have to add them to the aspects as before, because they are already declared in ObserverProtocol. Instead, we make ObservingOpen inherit from ObserverProtocol.

This in turn forces us to perform several adjustments. One is the conflict caused by two elements having the name Observer, which we again resolve by removing the import to java.util.Observer and making all references to this interface use the full compound name. The other is the creation of the definitions required by the abstract declarations in ObserverProtocol – the subjectChange pointcut and the updateObserver method. At this stage we start with a blank definition of updateObserver, while subjectChange is already created with the expression capturing the events of interest:

```
protected pointcut subjectChange(Subject subject):
   execution(void Subject+.open()) && this(subject);
protected void updateObserver(Subject s, Observer o) { }
```

At this intermediate stage, ObservingOpen's internal structure is rather confused, having reached its maximum intensity of smells (see Listing 44). Fortunately, modularisation puts us in a good position to start rationalising it.

```
import java.util.Observable;

public aspect ObservingOpen extends ObserverProtocol {
   protected pointcut subjectChange(Subject subject):
      execution(void Subject+.open()) && this(subject);
   protected void updateObserver(Subject s, Observer o) { }

   private interface BreakfastTaker {
      public void breakfastTime();
   }
   declare parents: (Bee || Hummingbird) implements BreakfastTaker;

   static class OpenNotifier extends Observable {
      private Flower _enclosing;
      private boolean alreadyOpen = false;
      public OpenNotifier(Flower flower) {
         _enclosing = flower;
      }
      public void notifyObservers() {
         if(_enclosing.isOpen() && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
      public void close() {
         this.alreadyOpen = false;
      }
   }
   static class OpenObserver implements java.util.Observer {
      private BreakfastTaker _enclosing;
      public OpenObserver(BreakfastTaker enclosing) {
         _enclosing = enclosing;
      }
      public void update(Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
   private OpenNotifier Flower.oNotify = new OpenNotifier(this);
   private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
   private OpenObserver Bee.openObsrv = new OpenObserver(this);

   public Observable Flower.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning : flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   pointcut flowerClose(Flower flower):
      execution(void close()) && this(flower);
   after(Flower flower): flowerClose(flower) {
      flower.oNotify.close();
   }

   public java.util.Observer Bee.openObserver() {
      return openObsrv;
   }
   public java.util.Observer Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

Listing 44: ObservingOpen as subaspect of ObserverProtocol

The code's first rationalisation step is to apply *Generalise Target Type with Marker Interface* (113). This entails assigning the roles to the participants by means of a 'declare parents':

```
public aspect ObservingOpen extends ObserverProtocol {
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;
   //...
```

We apply *Generalise Target Type with Marker Interface* (113) to the Flower type first. We replace all references to Flower with Subject, and again we use *Extend Marker Interface with Signature* (113) to extend Subject with the isOpen signature, which in turn forces us again to make the Flower.isOpen method public.

```
public class Flower {
   //...
   public boolean isOpen() {
      return isOpen;
   }

public aspect ObservingOpen extends ObserverProtocol {
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;

   protected pointcut subjectChange(Subject subject):
      execution(void Subject+.open()) && this(subject);

   protected void updateObserver(Subject s, Observer o) { }

   public abstract boolean Subject.isOpen();
```

Next, we replace references to Bee and Hummingbird with references to the Observer marker interface, which again enables us to replace references to the inner interface BreakfastTaker with references to Observer, and remove BreakfastTaker, as well as the 'declare parents' based on it. We also use *Extend Marker Interface with Signature* (113) again, this time to extend Observer with the breakfastTime signature.

```
   public OpenObserver(BreakfastTaker enclosing) {
      _enclosing = enclosing;
   }

   static class OpenObserver implements java.util.Observer {
      private BreakfastTaker _enclosing;
      public OpenObserver(BreakfastTaker enclosing) {
         _enclosing = enclosing;
      }
      public void update(Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
```

```
   public OpenObserver(BreakfastTaker enclosing) {
      _enclosing = enclosing;
   }
   public abstract void Observer.breakfastTime();
   //...
   static class OpenObserver implements java.util.Observer {
      private Observer _enclosing;
      public OpenObserver(Observer enclosing) {
         _enclosing = enclosing;
      }
      public void update(Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
```

As before, the replacements of Bee and Hummingbird also enable us to eliminate the duplication in the introductions of the field openObsrv and the method openObserver to Bee and Hummingbird. It is worth noting that passing 'this', which now refers to an instance of Observer, to the constructor of OpenObserver only resolves because we first replaced all references to BreakfastTaker with Observer.

```
private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
private OpenObserver Bee.openObsrv = new OpenObserver(this);
//...
public java.util.Observer Bee.openObserver() {
   return openObsrv;
}
public java.util.Observer Hummingbird.openObserver() {
   return openObsrv;
}
```

```
private OpenObserver Observer.openObsrv = new OpenObserver(this);
//...
public java.util.Observer Observer.openObserver() {
   return openObsrv;
}
```

This completes the application of *Generalise Target Type with Marker Interface* (113) to the Flower, Bee and Hummingbird types. From this point on, the participants are referred only in the role-assigning 'declare parents'. In the process, we noticed that the pointcuts flowerOpen and subjectChange are identical, so we removed flowerOpen, replacing it by subjectChange in the advice.

Listing 45 shows ObservingOpen at this point. Its internal structure again benefited from some tidying up. Further improvements demand that the existing implementation, still based on inner classes implementing the Observable/Observer protocol from java.util, be replaced by the implementation defined in ObserverProtocol. Only then, it becomes possible to remove the inner classes and the dependence on the Observable/Observer protocol from java.util.

```
import java.util.Observable;

public aspect ObservingOpen extends ObserverProtocol {
   protected pointcut subjectChange(Subject subject):
      execution(void Subject+.open()) && this(subject);
   protected void updateObserver(Subject s, Observer o) { }

   private interface BreakfastTaker {
      public void breakfastTime();
   }
   declare parents: (Bee || Hummingbird) implements BreakfastTaker;

   static class OpenNotifier extends Observable {
      private Flower _enclosing;
      private boolean alreadyOpen = false;
      public OpenNotifier(Flower flower) {
         _enclosing = flower;
      }
      public void notifyObservers() {
         if(_enclosing.isOpen() && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
      public void close() {
         this.alreadyOpen = false;
      }
   }
   static class OpenObserver implements java.util.Observer {
      private BreakfastTaker _enclosing;
      public OpenObserver(BreakfastTaker enclosing) {
         _enclosing = enclosing;
      }
      public void update(Observable ob, Object a) {
         _enclosing.breakfastTime();
      }
   }
   private OpenNotifier Flower.oNotify = new OpenNotifier(this);
   private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
   private OpenObserver Bee.openObsrv = new OpenObserver(this);

   public Observable Flower.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning : flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   pointcut flowerClose(Flower flower):
      execution(void close()) && this(flower);
   after(Flower flower): flowerClose(flower) {
      flower.oNotify.close();
   }

   public java.util.Observer Bee.openObserver() {
      return openObsrv;
   }
   public java.util.Observer Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

Listing 45: ObservingOpen after some tidying up.

An analysis of both implementations reveals a few hurdles. ObserverProtocol expects the events triggering the reactions of the observers to be represented by a *single* pointcut – subjectChange – but in this particular case, it is convenient to use *two*. In addition, this case requires that notification of all observers in the subject's list depend on the result of a test (if it is the first time that the event occurs). Only if the test succeeds are the subject's registered observers notified. This test relies on the alreadyOpen field of the OpenNotifier inner class. The same functionality can be obtained after moving that field to Flower, as an inter-type declaration. The problem is that the point where that test should be made is in place within ObserverProtocol, in the advice linked to the subjectChange pointcut:

```
   protected abstract pointcut subjectChange(Subject s);
   after(Subject s): subjectChange(s) {
      Iterator iter = getObservers(s).iterator();
      while ( iter.hasNext() ) {
         updateObserver(s, ((Observer)iter.next()));
      }
   }
```

This means that ObserverProtocol must be modified so that it can be used in this example. Since a subaspect cannot override the advice inherited from its superaspect, the advice itself will have to be pushed down from ObserverProtocol to ObservingOpen. There is also a piece of functionality missing in ObserverProtocol: the ability to clear all observers subscribing to a given subject. We address this last hurdle by adding to ObserverProtocol the following method:

```
public abstract aspect ObserverProtocol {
   //...
   public void clearObservers(Subject s) {
      getObservers(s).clear();
   }
```

Next, we add a new alreadyOpen field to Subject, as an inter-type declaration.

```
public aspect ObservingOpen extends ObserverProtocol {
   //...
   private boolean Subject.alreadyOpen = false;
```

We use *Push Down Advice* (133) on the advice triggered by subjectChange. ObservingOpen now has two pieces of advice on this pointcut, related to the old and new implementations respectively. Having two advice of the same kind on the same pointcut is clearly bad style, but in this case we tolerate it because the advice related to the old implementation is about to be removed. We now add to the new implementation the logic still missing, which relies on the alreadyOpen field introduced to Subject. This involves adapting the advice pulled down from ObserverProtocol according to the rules of this case:

```
public aspect ObservingOpen extends ObserverProtocol {
   //...
   protected pointcut subjectChange(Subject subject):
      execution(void Subject+.open()) && this(subject);
   after(Subject s): subjectChange(s) {
      Flower f = (Flower)s;
      if(f.isOpen() && !f.alreadyOpen) {
         Iterator iter = getObservers(s).iterator();
         while ( iter.hasNext() ) {
            updateObserver(s, ((Observer)iter.next()));
         }
      }
   }
```

We add the advice on the pointcut related to the execution of the close method of Flower

```
public aspect ObservingOpen extends ObserverProtocol {
   //...
   pointcut flowerClose(Subject flower):
      execution(void close()) && this(flower);

   after(Subject flower): flowerClose(flower) {
      if (subject instanceof Flower) {
         ((Flower)subject).alreadyOpen = false;
      }
   }
```

We finish this part by filling the aspect method updateObserver with the suitable logic:

```
public aspect ObservingOpen extends ObserverProtocol {
   //...
   protected void updateObserver(Subject s, Observer o) {
      o.breakfastTime();
      ((Flower)s).alreadyOpen = true;
   }
```

The new implementation is ready to be tested. However, we still do not have a test ready, because this time the changes impact on client code. This time we choose to create a new test rather than changing the existing one. This enables us to backtrack easily, in case something unexpected happens. Adding a new test requires a prior restructuring of the test class, by factoring out the initialisation code in the setUp method and cleanup code in the tearDown method (see Listing 46). Note that both the setUp and tearDown methods are called automatically by JUnit's TestCase class. This is the classic JUnit procedure for ensuring that the order of the tests is irrelevant and the tests do not interfere with each other.

```
public class TestObservedFlower extends TestCase {
   Flower f = new Flower();
   Bee ba = new Bee("A"),
       bb = new Bee("B");
   Hummingbird
      hx = new Hummingbird("X"),
      hy = new Hummingbird("Y");
   public void setUp() {
      f = new Flower();
      ba = new Bee("A");
      bb = new Bee("B");
      hx = new Hummingbird("X");
      hy = new Hummingbird("Y");
   }
   public void tearDown() {
      f = null;
      ba = null;
      bb = null;
      hx = null;
      hy = null;
      SpecificOutputCapture.clearResult();
   }
   private static String expectedOutput() {
      return //...
   }

   public void test() {
      //...
   }
   //...
}
```

Listing 46: Restructured version of test class.

We now add a new test (see Listing 47), which is a copy of the original test, but based on the new implementation. Note that the test string – returned by the expectedOutput method – is the same for both tests, which ensures that the behaviour being tested is the same in both cases. We also take the care to reverse the order with which the observers are registered, for same reason as before.

```
public class TestObservedFlower extends TestCase {
   //...
   public void test2() {
      ObservingOpen.aspectOf().addObserver(f, hy);
      ObservingOpen.aspectOf().addObserver(f, hx);
      ObservingOpen.aspectOf().addObserver(f, bb);
      ObservingOpen.aspectOf().addObserver(f, ba);

      f.closing().addObserver(ba.closeObserver());
      f.closing().addObserver(bb.closeObserver());
      f.closing().addObserver(hx.closeObserver());
      f.closing().addObserver(hy.closeObserver());
      // Hummingbird Y decides to sleep in:
      ObservingOpen.aspectOf().removeObserver(f, hy);

      // A change that interests observers:
      f.open();
      f.open(); // It's already open, no change.
      // Bee A doesn't want to go to bed:
      f.closing().deleteObserver(ba.closeObserver());

      f.close();
      f.close(); // It's already closed; no change
      ObservingOpen.aspectOf().clearObservers(f);

      f.open();
      f.close();
      Assert.assertEquals(expectedOutput(), SpecificOutputCapture.getResult());
   }
   //...
}
```

Listing 47: New test for the second implementation of ObservingOpen.

We compile and test: it passes. Again, both implementations can now be said to provide the same behaviour.

We now remove the old implementation and the original test. We could consider one last tidying to do: the extension of the Observer interface with the breakfastTime method is still in place. Though we could replace it with type conversions within the method where it is breakfastTime used, it is rather convenient in this case, because it is affecting two different and unrelated types. For this reason we choose to leave it there. See Listing 48 for the second final version of ObservingOpen.

We now perform a very similar sequence of steps to the second aspect (see Listing 49), only this time there was no need to make any more methods public. The test class also has two test methods temporarily, but in the end includes only the one shown in Listing 50.

```
import java.util.Iterator;

public aspect ObservingOpen extends ObserverProtocol {
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;

   private boolean Subject.alreadyOpen = false;
   public abstract void Observer.breakfastTime();

   protected pointcut subjectChange(Subject subject):
      execution(void Subject+.open()) && this(subject);
   after(Subject s): subjectChange(s) {
      Flower f = (Flower)s;
      if(f.isOpen() && !f.alreadyOpen) {
         Iterator iter = getObservers(s).iterator();
         while ( iter.hasNext() ) {
            updateObserver(s, ((Observer)iter.next()));
         }
      }
   }
   pointcut flowerClose(Subject flower):
      execution(void close()) && this(flower);
   after(Subject subject): flowerClose(subject) {
      if (subject instanceof Flower) {
         ((Flower)subject).alreadyOpen = false;
      }
   }

   protected void updateObserver(Subject s, Observer o) {
      o.breakfastTime();
      ((Flower)s).alreadyOpen = true;
   }
}
```

Listing 48: Second Final version of ObservingOpen.

```
import java.util.Iterator;

public aspect ObservingClose extends ObserverProtocol {
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;

   public abstract void Observer.bedtimeSleep();
   private boolean Subject.alreadyClosed = false;

   protected pointcut subjectChange(Subject subject):
      execution(void Subject+.close()) && this(subject);
   after(Subject subject): subjectChange(subject) {
      Flower f = (Flower)subject;
      if(!f.isOpen() && !f.alreadyClosed) {
         Iterator iter = getObservers(subject).iterator();
         while ( iter.hasNext() ) {
            updateObserver(subject, ((Observer)iter.next()));
         }
      }
   }
   pointcut flowerOpen(Subject flower):
      execution(void open()) && this(flower);
   after(Subject subject) returning : flowerOpen(subject) {
      if (subject instanceof Flower) {
         ((Flower)subject).alreadyClosed = false;
      }
   }
   protected void updateObserver(Subject s, Observer o) {
      o.bedtimeSleep();
      ((Flower)s).alreadyClosed = true;
   }
}
```

Listing 49: Final version of ObservingClose

```
   public void test() {
      ObservingOpen.aspectOf().addObserver(f, hy);
      ObservingOpen.aspectOf().addObserver(f, hx);
      ObservingOpen.aspectOf().addObserver(f, bb);
      ObservingOpen.aspectOf().addObserver(f, ba);

      ObservingClose.aspectOf().addObserver(f, hy);
      ObservingClose.aspectOf().addObserver(f, hx);
      ObservingClose.aspectOf().addObserver(f, bb);
      ObservingClose.aspectOf().addObserver(f, ba);

      // Hummingbird Y decides to sleep in:
      ObservingOpen.aspectOf().removeObserver(f, hy);

      // A change that interests observers:
      f.open();
      f.open(); // It's already open, no change.

      // Bee A doesn't want to go to bed:
      ObservingClose.aspectOf().removeObserver(f, ba);

      f.close();
      f.close(); // It's already closed; no change
      ObservingOpen.aspectOf().clearObservers(f);

      f.open();
      f.close();
      Assert.assertEquals(expectedOutput(), SpecificOutputCapture.getResult());
   }
```

Listing 50: Refactored version of the test for the Flower example.

7.8. Discussion

The refactoring process presented in this Chapter demonstrates that extractions based on inter-type declarations do not change the original design. They merely modularise it. The OOP paradigm led to a decentralised design that was still in place it was modularised within an aspect, though it was no longer suitable. Designs like this must be improved, if not downright replaced, after they are extracted to aspects. Yet, it still makes sense to first modularise the code, because improvements are easier to achieve when we can benefit from the advantages of modularity.

The refactoring example also shows how hard is to achieve reusable modules, even with AOP. The abstract, “reusable” aspect for the Observer pattern had to undergo invasive changes just to be used in Eckel’s simple example. We nevertheless believe that this aspect has the potential to be further developed in order to encompass a wider variety of situations, though that will almost certainly come at the cost of making its design more complex.

Chapter 8. Related Work

This chapter highlights contributions that bear a relation to our work or that potentially have an impact on its future directions. Each section of this chapter covers a different angle, discussing works related to that angle. In section 8.1 we refer to an overview that covers the subject of this thesis. In section 8.2, we refer to a work focusing the same case study we describe in Chapter 4. Our work relates to the manual application of refactorings, which pose particular problems. In section 8.3, we refer to works proposing guidelines that tackle such problems. Style guidelines bear a close relationship to our work, as refactorings and code smells are just one of various possible ways way to express them (we also argue they are particularly effective at that). In section 8.4, we cover the subject of general style guidelines. We dedicate the following two sections to the specific cases of refactorings and code smells: in section 8.5, we survey proposals of AO refactorings by other authors, and in 8.6 we discuss code smells. In section 8.7, we refer to a work related to proving the preservation of behaviour in AO refactorings whose continuation promises to have an impact on ours. Finally, in section 8.8 we mention potential problems and pitfalls of AOP whose solutions may include style rules.

8.1. General Overview

Van Deursen et al. [36] provide an overview of the state of art in the area of aspect mining and refactoring. Though their main concern seems to be tools for the automatic detection of aspects, they also mention several open questions about refactoring to aspects. These include:

- "How can existing code smells be used to identify candidate refactorings?"
- "How can the introduction of aspects be described in terms of a catalogue of new refactorings?"

In this thesis, we contribute to answering these two questions.

8.2. Assessing the GoF Pattern Implementations Code

In [49], Garcia et al provide a quantitative study comparing the OO and AO versions of the code examples by Hannemann and Kiczales [60], which comprises the case study presented in Chapter 4. Their study is based on well-known software engineering attributes such as separation of concerns, coupling, cohesion and size. They performed minor modifications on the code examples, to ensure the equivalence between the two versions. In addition, they increased the number of participant classes playing pattern roles. Their study was based on both the original code and their extended versions.

Their findings generally confirm the claims by Hannemann and Kiczales [60] that AspectJ implementations improve on those in Java, but there are qualifications. In particular, their study indicates that only 4 (rather than 12) of the patterns yield reusable AspectJ implementations: Mediator, Observer, Composite, and Visitor. Garcia et al. remark that in general, reusable elements lead to a decrease in the programming effort by requiring fewer operations and lines of code to be written. However, some of the metrics they employed yielded worse measurements for the AspectJ implementations of some patterns – Flyweight, Command, Chain of Responsibility, Memento, Prototype, Singleton and

Strategy – than in the respective OO implementations. Garcia et al. remark that these patterns are very simple and do not enable a reasonable degree of reuse.

Garcia et al. also complain about the complexity in using some of the reusable aspects, which translates in some unfavourable measurements in some of patterns – Memento, Strategy and Chain of Responsibility. Interestingly, they do not include Composite in this group, which we single out in our own complaints about complexity (see 4.3.2). One possible explanation for the discrepancy is the fact that our views are based solely on the original examples, while the more complex versions created by Garcia et al. may provide a different picture.

8.3. Guidelines on How to Approach the Source Code

The team in charge of developing AJDT [169] is working on tool support for object-to-aspect refactorings. In [28] Clement et al. present their plans on that front and describe how existing functionality provided for the eclipse IDE [179] – such as the planned crosscutting "diff" view – could be used to assist in extracting advices from a code base. In addition, Clement et al. provide some general guidelines for performing an extraction of an aspect from its code base to an aspect. One particularly useful tip, which we include in our refactorings, is to use the AspectJ 'declare warning' statement to capture all the scattered places in the code where the tangled logic to be extracted resides.

In [91] Laddad prescribes several guidelines to ensure AO refactorings for concern extraction are applied in a safe way. These involve the creation of a first version of the pointcut, based on a case-by-case enumeration of the interesting joinpoints, followed by its replacement with a semantically more meaningful pointcut, based on wildcards. Laddad also proposes a mechanism based on AspectJ's 'declare error' mechanism to verify whether two different pointcut expressions capture exactly the same set of joinpoints (described in section 6.2). In addition, Laddad recommends that in the beginning, aspects be developed with a restricted scope, often affecting the methods of a single class, in order to make it simpler to test their impact on the base code. Only afterwards should the scope of the aspect widen, when its functionality is already tested with the restricted case. In other words, Laddad expresses the view that pointcuts should be *bounded*, in the sense proposed by McEachen and Alexander [107]: "bounded pointcuts are pointcuts defined such that they only match join points from packages and classes present in the original environment that the aspect was specifically developed and tested for. *Unbounded pointcuts* can potentially match join points beyond the original environment."

Considering that at present there is no adequate tool support for AO refactorings, and that aspects can potentially impact a large number of joinpoints across an entire system, procedures such as those proposed by Laddad are essential to any refactoring process targeting non-trivial systems.

8.4. General Style Rules

In the last few years many projects were undertaken to assess the applicability of AO techniques [159][62] to various domains, covering various areas of applicability, including image processing [109], exception detection and handling [98], web-based applications [78] and distribution and persistence [146]. Few authors of these projects draw conclusions about the appropriate AO style. One of the few works that partially focus on style issues is the one by Kersten and Murphy [78], who explored a case study of the use of AOP to the (re)development of ATLAS, a web-based learning environment initially developed in C++. Kersten and Murphy report that at the start of the project they did not pay much attention

to the style in which they used the constructs of AspectJ. As the application grew to be over 50 classes and aspects, it became increasingly difficult to understand and test the classes, because of the way in which they were using aspects was making it hard to reason about how all the code fit together. To help manage the complexity, they began to constrain and stylise the aspect code. In the process, they derived the following three aspect style rules:

- *Exceptions introduced by a weave must be handled in the code comprising the weave.* This rule means that if the code being introduced into a method can raise an exception, it should be wrapped in a try block that handled that exception.
- *Advise weaves must maintain the pre- and post conditions of a method.* This rule, together with the first, ensures that an aspect does not change the contract between a class and its clients. This is to ensure that classes affected by aspects continue to fit in the existing class structure.
- *Before advise weaves must not include a return statement.* This rule was meant to ensure that the code defined in the main body of the method is always run. This rule is now obsolete: in the present version of AspectJ, a return within a before advice (assuming the advised joinpoint is related to the execution of methods) does not prevent the method from running.

We find the rules to be sound, but they only scratch the surface of the style space. The versions of AspectJ used in the study were very early (versions 0.2.0beta4 through beta10). Consequently, the authors do not cover many issues arising from the more mature AspectJ. For instance, Laddad provides a more elaborate description of how to handle exceptions in the *Exception Introduction* pattern he presents in his book ([90], p.260), and further elaborates on the subject in [91]. The work by Kersten and Murphy was focused on building a new architecture (there was an initial version of ATLAS, developed in C++), while our work focuses on finding novel code transformations.

8.5. Proposals for Aspect-Oriented Refactorings

Iwamoto and Zhao announced in [73] their intention to build a catalogue of AO refactorings. They present a catalogue of 24 refactorings, but the information provided about them is limited to the names of the refactorings. The refactorings we document in this thesis include a description of the situations where the refactoring applies, mention of preconditions, detailed mechanics and code examples.

Hanenberg et al. [58] also propose three AO refactorings – *Extract Advice*, *Extract Introduction* and *Separate Pointcut*. Their *Extract Advice* corresponds to our *Extract Fragment into Advice* refactoring (94). The collection of refactorings presented in this thesis goes deeper in exploring the refactoring space, providing more detail and tackling issues such as the tidying up of the internal structure of aspects resulting from extraction processes.

We do not subscribe the recommendation by Hanenberg et al., proposed in their *Extract Advice* refactoring, to use around advice in the general case. We think that in cases where either before or after advice can be used, these should be used in preference to around, because it makes the scope of the advice easier to perceive at a first look at the code. The around advice is also more powerful than is often needed. In the case of code using around advice without a strict need for it, we envision refactorings such as *Change Around Advice to Before* and *Change Around Advice to After Returning*. Their proposed *Extract Introduction* refactoring corresponds to our *Move Field from Class to Inter-type* (104) and *Move Method from Class to Inter-type* (106) refactorings, which provide more detail. *Separate Pointcut* relates to

evolution of pointcuts and has no correspondence in our collection. This refactoring argues that, just as it is beneficial to organise our systems using small methods with meaningful names, we should do the same with pointcuts.

Rura [139] documented aspect aware versions of some of the traditional OO refactorings, using giving new preconditions and transformation steps. Rura also contributes with a collection of very low-level aspect-oriented refactorings, for instance the creation of empty aspects, methods, pointcuts, and deletion of unreferenced classes and class members. Rura also contributes with a few refactorings for manipulating members across classes and aspects, which partially overlap with the extraction refactorings we document in section 6.3. All descriptions are done in a style akin to that of Opdyke [125], which is succinct and does not include code examples.

Laddad presents a collection of refactorings [91] with a significant utility value, particularly to developers of J2EE applications. The refactorings vary widely in both level and scope of applicability, including generally applicable refactorings like *Extract Interface Implementation, Extract Method Calls*, and *Replace Override with Advice*, but also concern-specific refactorings such as *Extract Concurrency Control* and *Extract Contract Enforcemement*. In addition, some refactorings belong to the category of “refactoring to patterns” such as those presented by Kerievsky [76] – *Extract Worker Object Creation* and *Replace Argument Trickle by Wormhole*. These two refactorings are based on two of the design patterns presented by Laddad in [90] – *Worker Object Creation* ([90], p.247) and *Wormhole* ([90], p.256) respectively. The *Extract Exception Handling* refactoring as presented in [91] goes towards a variant implementation of the *Exception Introduction* pattern ([90], p.260).

We believe programmers would benefit if Laddad’s refactorings were presented in the same format as used by Fowler et al. [47] and Kerievsky [76], and which we use as well (some refactorings are presented with only a mention of its name and a brief motivating paragraph). A mechanics section would be particularly beneficial, having proved very useful as a checklist, and to lead developers through the safest sequences of steps, in preference to riskier or less convenient ones. The important step-by-step guidelines proposed by Laddad for creating a new aspect and subsequently evolving it are included in the code example illustrating the use of *Extract Method Calls*, but not in several other refactorings to which they also apply (Laddad places some reminders). A mechanics section would make that part process clearer, and would clarify the relations between refactorings.

We noticed that several of Laddad’s refactorings, namely the problem-specific ones, can be decomposed into simpler, lower level steps – always an important thing with (manual) refactoring. During our work on the mechanics of the refactorings that are documented in Chapter 6, we focused on the minute details of the refactoring process, enabling us to improve their characterisation. In some cases, this led us to decompose the refactoring under study into several smaller steps. For instance, *Split Abstract Class into Aspect and Interface* (109) and *Change Abstract Class to Interface* (88) were initially conceived as a single refactoring. The pair comprising *Extract Inner Class to Standalone* (99) and *Inline Class within Aspect* (102) was likewise meant to perform what is a single logical transformation: moving an inner class from a class to an aspect. We believe similar benefits can be obtained by similarly approaching Laddad’s refactorings – some of the resulting lower level steps would correspond to existing steps, while others would possibly yield new refactorings.

8.6. Proposals for Aspect-Oriented Code Smells

Hanenberg et al. [58] do not elaborate on code smells, but we can infer from *Separate Pointcut* that these authors consider anonymous pointcuts to be a code smell.

Laddad [91] does not pinpoint the code smells that his refactorings are supposed to remove. We think that the material presented by Laddad has the potential to throw new light on existing OO code smells or to yield new ones. For instance, his *Extract Method Calls* and *Replace Argument Trickle by Wormhole* refactorings respectively suggest the *Scattered Method Calls* and *Argument Trickle* smells. Further research is required to discover latent smells and assess their feasibility and applicability.

Tonella and Ceccato [153] base their work on the assumption that interfaces are often (not necessarily always) related to concerns other than the one pertaining to the system's main decomposition. This is an *Interface Implementation* smell, though the authors do not name it this way. They provide specific guidelines for when an interface implementation is a symptom of a latent aspect and present a tool for mining and extracting aspects based on these criteria, and report on experimental results. These extractions are also covered by the refactorings we document in this thesis. The authors also point out various issues that can arise in a typical extraction of an interface implementation into an aspect. Our refactorings prescribe procedures to deal with all these issues.

To our knowledge, no work besides ours deals with the potentially bad internal structure of aspects resulting from extraction processes. With the exception of the work by Tonella and Ceccato [153], we do not have knowledge of any other work explicitly covering the issue of AO code smells.

8.7. Proving the Preservation of Behaviour

In [29] and [30] Cole and Borba propose programming laws from which refactorings for AspectJ can be derived. The authors focus on the use of their laws to derive existing refactorings such as those proposed in [91], [58] and [73], and describe two case studies in which the laws were tested, comprising the extraction of concurrency control and distribution respectively. Many, though not all, of the laws relate to the extraction of crosscutting concerns to aspects, and therefore there is some overlap between the refactorings they derive and our own extraction refactorings (see section 6.3). However, their main emphasis is to provide proofs that the transformations are behaviour preserving, while we focus on covering new ground in the refactoring space. Nevertheless, the authors remark that extraction procedure for the second case study is generalisable, because its implementation of distribution is commonly used, and claim that it is possible to derive a concern-specific *Extract Distribution* refactoring.

8.8. Potential Pitfalls

Though they do not use this name, several authors [73][58][154][162] call into attention what we call the *fragile base code problem* (see beginning of Chapter 6), in some cases illustrating it with some code examples. These authors conclude that existing OO refactorings [47] cannot be applied to code bases with aspects. We believe these problems can be ameliorated if adequate procedures are followed [91]. These include adoption of an appropriate style for programming and evolving aspect constructs, particularly pointcuts. Such a style also entails following the rule, proposed by Laddad, that aspects be placed as near the code they affect, in order to guarantee as much as possible that programmers are aware of potential problems when changes are made to the base code or the aspects.

To tackle the fragile base code problem, Hanenberg et al. [58] propose *aspect-aware* refactorings – refactorings that take into account the presence of aspects and preserve behaviour by updating any pointcuts that may be affected by the transformation – and propose a set of enabling conditions to preserve the observable behaviour. By the author's

admission, these conditions must be automatically verified by an aspect-aware tool, as the manual verification is an exhausting task, even in small systems. Hanenberg et al. announce a tool providing a subset of the functionality they deem desirable.

McEachen and Alexander [107] call into attention a series of problems that can arise in situations of a software system being composed of classes and aspects that originate from two or more independent development teams. In case components that include aspects are distributed without source code, various situations can arise in which the lack of awareness of unknown or undocumented assumptions can lead to negative phenomena, including compilers errors, unexpected behaviour and inconsistent behaviour. These problems are reminiscent of the fragile base class problem [114], and McEachen and Alexander do indeed reference that work. It is certain that the solutions to the problems mentioned by McEachen and Alexander require tools and techniques not presently available, but part of the solution also involves style rules. One case that can be discerned already is the case of unbounded pointcuts. Unbounded pointcuts risk weaving more joinpoints than those initially intended, and therefore McEachen and Alexander recommend that unbounded pointcuts be avoided, unless compelling reasons exist for them to being this way.

Chapter 9. Conclusions

This Ph.D. thesis contributes to characterise a new style appropriate for aspect-oriented source code, using refactorings and code smells as a means to express it. In this Chapter, we review our central results and primary contributions, and propose several ideas for future research.

9.1. Contributions

The underlying goal of this thesis is to contribute to the characterisation of a new programming style, appropriate for aspect-oriented source code. We aim to build a collection of refactorings and code smells capable of expressing part of such a style, in the same way the refactorings and code smells documented by Fowler et al. successfully represent a notion of good style for OO source code. Though various authors proposed various aspect-oriented refactorings [73][58][91], none pinpointed the code smells those refactorings are supposed to remove.

The primary contributions of this thesis are the collections of novel refactorings for the AspectJ programming language. In addition, we propose various code smells that provide the broad motivation for using the refactorings. We examine the traditional OO smells and propose that some of them be used as indicators of the presence of crosscutting concerns in OO source code (see section 5.1). We also define two novel code smells – *Double Personality* (78) and *Abstract Class* (80) – to also help to detect crosscutting concerns in existing OO code. In addition, we propose the *Aspect Laziness* (80) smell, which is specific to aspects. All code smells are presented in Chapter 5.

The refactorings are documented in Chapter 6, in a style and level of detail akin to that used in Fowler's book [47], including a motivation section and code examples. The descriptions contribute to establish the context in which use of the refactoring can be beneficial. They are divided into three groups, plus one special case. The three main groups of refactorings comprise:

1. Ten refactorings for the extraction of crosscutting concerns from Java code bases to aspects.
2. Six refactorings for improving the internals of aspects, particularly aspects resulting from extractions performed with the refactorings of the previous group.
3. Eleven refactorings to deal with the extraction of common code from multiple aspects and for moving aspect-specific constructs between superaspects and subaspects.

The most original refactorings documented in this thesis are the ones from the second group. We characterise the potential bad internal structure of aspects that result from extraction processes through an aspect-specific smell – *Aspect Laziness* (80). The refactorings from the second group can remove this smell from existing aspects.

In addition to the three main groups of refactorings, we defined one refactoring to deal with the separation of concerns in the signature of constructors that are part of published interfaces.

The refactorings are validated through a refactoring example in which seventeen of the refactorings are used. The refactorings used come from all the three main groups mentioned above, and include all the refactorings from the second group.

9.2. Publications

At the moment of finishing this thesis, the contributions are presented/documented in the following publications:

- [116] and [121] state the aims of this Ph.D. project. [121] also presents a few early findings.
- [118] presents the first case study and documents five refactorings that were derived from that study.
- [119] presents a short analysis of the code examples that comprised our second case study in the light of ease of use and reusability.
- [115] is a technical report documenting all twenty-eight refactorings, in a style and format similar to the ones used in [47] and [76]. We chose to document this part of our work through a technical report because we concluded that it would be extremely tricky to document refactorings in conference proceedings. The problems are due to space constraints and the fact that small sets of refactorings are not likely to be considered a substantial enough contribution to be accepted by the review boards of prestigious international conferences.
- [123] presents most of the refactorings, reviews the traditional OO code smells in light of AOP, proposes several novel aspect-oriented code smells, including one that is specific to aspects. It presents considerations of various orders, including the effects of crosscutting in the design of existing OO systems. It also surveys related work and proposes several new directions of research in this field.
- [120] presents a refactoring process of a Java code base into an AspectJ equivalent, using seventeen of the refactorings documented in [118]. It is complemented with an eclipse project, available online, presenting dozens of complete snapshots of the code being refactored.
- [122] is a position paper for a workshop expressing the view that research specifically targeting issues of style play an important role in the current state-of-the-art in AO refactoring.

9.3. Future Research

In this section, we put forth various ideas for continuing and/or extending the work described in this thesis.

9.3.1 Maturing the Refactorings

There is scope for maturing the refactorings proposed in this thesis. This entails testing the refactorings with different code bases, particularly larger and more complex ones.

We consider the possibility that the composite refactorings such as *Extract Feature into Aspect* (90) and *Tidy Up Internal Aspect Structure* (128) will evolve to give origin to various refactorings in the conventional sense, or to be turned into an introductory text to a group of related refactorings.

9.3.2 Pinpointing Other Code Smells

In addition to the traditional OO smells we analysed in terms of AOP in Chapter 5, there are a few other smells we believe can be useful in detecting crosscutting concerns, but which we did not sufficiently explore to pinpoint suitable refactorings to remove them. We single out *Parallel Inheritance Hierarchies* ([47], p.83) and *Combinatorial Explosion* ([76], p.45 and [158] p.109) may be indicative of the presence of crosscutting concerns in some cases. *Combinatorial Explosion*, as proposed by Wake [158] is a variant of *Parallel Inheritance Hierarchies* ([47], p.83). We think it is worth investigating whether these smells could be considered symptoms of the *tyranny of the dominant decomposition* [150] in some cases.

Besides existing OO smells, there are many latent aspect-specific smells waiting to be discovered. For instance, privileged aspects: the rationale for avoiding them is the same as for avoiding the use of public data. As Colyer and Clement remark in [31], aspect privilege confers the general privilege to see any private state anywhere, while one often wishes to express privilege with respect to a single class or a restricted set of classes. Presently, this is not possible with AspectJ. Unfortunately, privileged aspect may be unavoidable in cases affecting multiple packages and in which the aspect needs access to non-public (e.g. protected and package-protected) data. Refactoring the affected code bases to expose the non-public data is one alternative. We need to study uses of privileged aspects to assess whether common patterns can be found, and pinpoint refactorings that tackle this issue.

9.3.3 Developing Other Refactoring Ideas

We detected a few latent refactorings in the material from our case studies. Next, follow a few examples of ideas for refactorings which look promising but which were not yet fully explored:

- *Replace Throws with Declare Error* – many existing instances of Java code throw an exception upon detection of illegal situations. Some of these situations can be specified by statically determinable pointcuts, in which case it is more effective to replace them with a 'declare error' clause.
- *Remove Signatures from Inner Interface* – As a rule, marker interfaces do not declare operations, so it is worth exploring a refactoring to remove the operations declared by an inlined interface.
- *Replace Downcast with Interface Extension* – We proposed *Extend Marker Interface with Signature* (113) to resolve dependencies to concrete types caused by calls to type-specific methods. This idea can be taken further by completely removing dependencies on a type, namely type casts, to the point of removing the import of the type.

More refactoring experiments are likely to yield other refactorings. Further experiments should target code bases with different characteristics than the ones used for this thesis. For instance, the case studies used would not be appropriate to develop refactorings to deal with exceptions – the first code base (Chapter 3) did not use them extensively enough, and the code examples from the second case study (Chapter 4) did not use them at all.

In addition, there are many possible variants to the refactorings proposed in this thesis. One example is to extract common code from multiple, similar aspects through an *Extract Subaspect* refactoring instead of an *Extract Superaspect* (129).

9.3.4 Covering Other Language Characteristics

The refactorings we derived are the result from two specific case studies, where not every possible AspectJ construct is used. New research should cover the remaining aspect constructs, as well as the interactions between these and traditional Java constructs. We next mention two subjects.

Non Singleton Aspect Association

Our work so far concentrated on singleton aspects. In future, we expect to cover other kinds of aspect association in order to obtain a clearer idea of the advantages and disadvantages of non-singleton aspects, e.g., when should they be preferred and what refactorings should be used to transform singleton aspects.

Pointcut style

At present, refactorings and code smells specifically targeting pointcuts are still a largely unexplored area. AspectJ's pointcut protocol comprises a rich language for quantification and is likely to yield an equally rich pattern language for refactoring pointcut expressions, as well as their interaction with advice. Laddad gives some suggestions on how to evolve pointcuts [91] but further research is needed on the adequate use of pointcut designators (e.g. smells to expose inadequately coded pointcuts), and how best to evolve pointcut expressions.

9.3.5 Refactorings for Restructuring the Remaining Base Code

The refactorings documented in Chapter 6 cover the restructuring of the code *within* aspects resulting from the extraction of crosscutting concerns, taking advantage of the newfound modularisation. We did not focus our work on the impact of such extractions on the remaining code base. Since many designs become more complex to take into account crosscutting effects, it is likely that in some cases the extraction of crosscutting concerns into aspects will provide the opportunity to further simplify the structure of the remaining code base. Further research is required to find patterns in refactored OO code bases and what post-extraction refactorings would be desirable.

9.3.6 Refactorings to Cope with Published Interfaces

Refactoring legacy code entails dealing with published interfaces, i.e. interfaces used by clients that developers cannot, or are not at liberty to change. Occasionally the tangling resulting from the presence of crosscutting concerns is present in the signatures, in which case it cannot be readily removed. In such cases, developers have the option to refactor *towards* rather than *to* a goal, while a deprecation policy is pursued. We partially dealt with that issue when we characterised the refactoring *Partition Constructor Signature* (137). We did not continue our work in that direction after we abandoned the first case study, but we deem this subject worthy of further research.

9.3.7 Developing Opposite Refactorings

In this thesis, we did not specifically aim to provide opposites for each of the proposed refactorings. Instead, we preferred to focus on extending the space of known refactorings. Nevertheless, opposites are important to enable developers to backtrack, whenever they find out they took a wrong turn, and because opposites are sometimes useful in their own right (e.g. pull up vs. push down refactorings).

9.3.8 Updating to New Language Versions

During the final months when this thesis was being written, important developments for the AspectJ language were under way. These promise to have an impact on some of the constructs of AspectJ. One is the update of AspectJ to attain compatibility with the latest version of Java (Java 1.5, codenamed "Tiger" [5]). Java 1.5 includes several new important features, including generics and metadata. In addition, the merge of AspectJ and Aspectwerkz [175] was announced [171]. These developments will yield a new version of the language called AspectJ 5 [173]. Though AspectJ 5 is likely to keep a backwards compatibility with most characteristics of the preceding version, this may have an impact on notions of good style. The rationale is the same as presented in Chapter 1: AspectJ 5 promises to achieve some effects better than the present version, and therefore the previous ways of doing things should be considered bad style. Therefore, the advent of AspectJ 5 may motivate new code smells and the corresponding refactorings.

9.3.9 Comparing AspectJ with AHEAD

The adoption of the AHEAD tool for the continuing development of WorkSCo, the first case study, was one of the primary motives for abandoning it in preference of the code examples presented in Chapter 4. However, the availability of a code base based on AHEAD opens the way to make some interesting comparisons between AHEAD and AspectJ.

This would possibly entail undertaking something close to the original aim for the first case study, to develop an AspectJ version of WorkSCo. This task would provide the opportunity to assess the relative strengths and limitations of both tools. One hypothesis that was contemplated when AHEAD was adopted, but not pursued, was to find out whether AspectJ could emulate AHEAD (and vice versa). Our experience with WorkSCo (see Chapter 3) suggests that the answer is negative, but it remains to be confirmed, and to find out exactly why and to what extent.

Appendix A
Refactorings Documented by Fowler

Add Parameter	Pull Up Constructor Body
Change Bidirectional Association to Unidirectional	Pull Up Field
Change Reference to Value	Pull Up Method
Change Unidirectional Association to Bidirectional	Push Down Field
Change Value to Reference	Push Down Method
Collapse Hierarchy	Remove Assignments to Parameters
Consolidate Conditional Expression	Remove Control Flag
Consolidate Duplicate Conditional Fragments	Remove Middle Man
Convert Procedural Design to Objects	Remove Parameter
Decompose Conditional	Remove Setting Method
Duplicate Observed Data	Rename Method
Encapsulate Collection	Replace Array with Object
Encapsulate Downcast	Replace Conditional with Polymorphism
Encapsulate Field	Replace Constructor with Factory Method
Extract Class	Replace Data Value with Object
Extract Hierarchy	Replace Delegation with Inheritance
Extract Interface	Replace Error Code with Exception
Extract Method	Replace Exception with Test
Extract Subclass	Replace Inheritance with Delegation
Extract Superclass	Replace Magic Number with Symbolic Constant
Form Template Method	Replace Method with Method Object
Hide Delegate	Replace Nested Conditional with Guard Clauses
Hide Method	Replace Parameter with Explicit Methods
Inline Class	Replace Parameter with Method
Inline Method	Replace Record with Data Class
Inline Temp	Replace Subclass with Fields
Introduce Assertion	Replace Temp with Query
Introduce Explaining Variable	Replace Type Code with Class
Introduce Foreign Method	Replace Type Code with State/Strategy
Introduce Local Extension	Replace Type Code with Subclasses
Introduce Null Object	Self Encapsulate Field
Introduce Parameter Object	Separate Domain from Presentation
Move Field	Separate Query from Modifier
Move Method	Split Temporary Variable
Parameterise Method	Substitute Algorithm
Preserve Whole Object	Tease Apart Inheritance

Appendix B
Code smells described by Beck and Fowler

Smell	Common Refactorings
Alternative Classes with Different Interfaces	Rename Method, Move Method
Comments	Extract Method, Introduce Assertion
Data Class	Move Method, Encapsulate Field, Encapsulate Collection
Data Clumps	Extract Class, Introduce Parameter Object, Preserve Whole Object
Divergent Change	Extract Class
Duplicated Code	Extract Method, Extract Class, Pull Up Method, Form Template Method
Feature Envy	Move Method, Move Field, Extract Method
Inappropriate Intimacy	Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation, Hide Delegate
Incomplete Library Class	Introduce Foreign Method, Introduce Local Extension
Large Class	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Lazy Class	Inline Class, Colapse Hierarchy
Long Method	Extract Method, Replace Temp with Query, Replace Method with Method Object, Decompose Conditional
Long Parameter List	Replace Parameter with Object, Introduce Parameter Object, Preserve Whole Object
Message Chains	Hide Delegate
Middle Man	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Parallel Inheritance Hierarchies	Move Method, Move Field
Primitive Obsession	Replace Data Value with Object, Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy
Refused Bequest	Replace Inheritance with Delegation
Shotgun Surgery	Move Method, Move Field, Inline Class
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method
Switch Statements	Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object
Temporary Field	Extract Class, Introduce Null Object

Appendix C
Additional refactorings from www.refactoring.com

Convert Dynamic to Static Construction	Refactor Architecture by Tiers
Convert Static to Dynamic Construction	Remove Double Negative
Extract Package	Replace Assignement with Initialization
Hide presentation tier-specific details from the business tier	Replace Conditional with Visitor
Introduce A Controller	Replace Iteration with Recursion
Introduce Business Delegate	Replace Recursion with Iteration
Introduce Synchronizer Token	Replace Static Variable with Parameter
Localize Disparate Logic	Reverse Conditional
Merge Session Beans	Split Loop
Move Business Logic to Session	Use a Connection Tool
Move Class	Wrap entities with session
Reduce Scope of Variable	

Appendix D
Refactorings to patterns by Kerievsky

Chain Constructors	Move Creation Knowledge to Factory
Compose Method	Move Embellishment to Decorator
Encapsulate Classes with Factory	Replace Conditional Dispatcher with Command
Encapsulate Composite with Builder	Replace Conditional Logic Strategy
Extract Adapter	Replace Constructors with Creation Methods
Extract Composite	Replace Hard-Coded Notifications with Observer
Extract Parameter	Replace Implicit Language with Interpreter
Form Template Method	Replace Implicit Tree with Composite
Inline Singleton	Replace One/Many Distinctions with Composite
Introduce Null Object	Replace State-Altering Conditionals with State
Introduce Polymorphic Creation with Factory Method	Replace Type Code with Class
Limit Instantiation with Singleton	Unify Interfaces
Move Accumulation to Collecting Parameter	Unify Interfaces with Adapter
Move Accumulation to Visitor	

Appendix E
Code smells described by Kerievsky

Smell	Refactorings
Alternative Classes with Different Interfaces	Unify Interfaces with Adapter
Combinatorial Explosion	Replace Implicit Language with Interpreter
Conditional Complexity	Replace Conditional Logic with Strategy Move Embellishment to Decorator Replace State-Altering Conditionals with State Introduce Null Object
Duplicated Code	Form Template Method Introduce Polymorphic Creation with Factory Method Chain Constructors Replace One/Many Distinctions with Composite Extract Composite Unify Interfaces with Adapter Introduce Null Object
Indecent Exposure	Encapsulate Classes with Factory
Large Class	Replace Conditional Dispatcher with Command Replace State-Altering Conditionals with State Replace Implicit Language with Interpreter
Lazy Class	Inline Singleton
Long Method	Compose Method Move Accumulation to Collecting Parameter Replace Conditional Dispatcher with Command Move Accumulation to Visitor Replace Conditional Logic with Strategy
Oddball Solution	Unify Interfaces with Adapter
Primitive Obsession	Replace Type Code with Class Replace State-Altering Conditionals with State Replace Conditional Logic with Strategy Replace Implicit Tree with Composite Replace Implicit Language with Interpreter Move Embellishment to Decorator Encapsulate Composite with Builder
Solution Sprawl	Move Creation Knowledge to Factory
Switch Statements	Replace Conditional Dispatcher with Command Move Accumulation to Visitor

Appendix F
Code smells and symptoms described by Wake

Symptoms	Smell
Duplication	
Methods and/or classes have similar behaviour but different names	Alternative classes with Different Interfaces
Similar code	Duplicated Code
Code with similar effects	
Names	
Two different names for the same thing	Inconsistent Names
Name is compound word, including a type name	Type Embedded in Name
Hungarian notation (type indicator used as prefix)	
Variable named after type rather than intent	
One- or two-character name	Uncommunicative Name
Name without vowels	
Numbered variables	
Odd abbreviations	
Name that doesn't reflect its real use or meaning	
Code within a Method	
// or /* */	Comments
Complex expression using &&, ||, !	Complicated Boolean Expression
Large number of lines in method	Long Method
Large number of parameters	Long Parameter List
Constant embedded in code	Magic Number
Comparison against null	Null Check
Complex if statement	Special Case
Conditional test before body of code	
switch keyword used	Simulated Inheritance (Switch Statement)
Several if statements in a row (comparing the same item)	
Use of instanceof	
Class	
Large number of instance variables in class	Large Class
Large number of methods in class	
Large number of lines in class	
Complexity	
Variable, parameter, field, code fragment, method, or class never referenced	Dead Code (May also be Specultive Generailty)
Code more general/complicated than it needs to be	Speculative Generailty

Symptoms	Smell
Data	
Public fields (or just setters and getters), but little behaviour in class	Data Class
Same 2-3 items occur together in classes and parameter lists	Data Clump
Fields named with similar substrings	
Allternating declarations of fields and methods	
Primitive types (int, string, etc.)	Primitive Obsession
Constants and enumerations	
String constants for field names	
Fields are sometimes set, sometimes null	Temporary Field
Inheritance	
Subclass is too tied to parent's data or methods	Inappropriate Intimacy (Subclass Form)
Class has little code in it	Lazy Class
Subclass throws exception on method the parent requires	Refused Bequest
Inherited method doesn't work	
Clients refer to subclass but never hold reference to the parent class	
Subclass isn't (conceptually, behaviorally) an example of the parent class	
Responsibility	
Class manipulates another class's data	Feature envy
Class relies too much on how another class works	Inappropriate Intimacy (General Form)
Chain of calls: a.b().c().d()	Message Chain
Delegation: f() { delegate.f(); }	Middle Man
Accomodating Change	
Introducing new class requires multiple classes at various points in its hierarchy	Combinatorial Explosion
Each level of hierarchy deals with a different attribute	
Same class changes for different reasons	Divergent change
Adding a subclass in one hierarchy requires corresponding subclasses in other hierarchies	Parallel Inheritance Hierarchies
Subclasses in two separate hierarchies use the same prefixes	
Multiple classes must change for a single decision	Shotgun Surgery
Working with Libraries	
Library doesn't have a feature you need	Incomplete Library Class

Appendix G – AspectJ Quick Reference

Aspects *at top-level (or static in types)*

aspect *A* { … }
defines the aspect *A*
privileged aspect *A* { … }
A can access private fields and methods
aspect *A* **extends** *B* **implements** *I, J* { … }
B is a class or abstract aspect, *I* and *J* are interfaces
aspect *A* **percflow**(*call(void Foo.m())*) { … }
an instance of *A* is instantiated for every control flow through calls to *m()*

general form:
[**privileged**] [*Modifiers*] **aspect** *Id*
[**extends** *Type*] [**implements** *TypeList*]
[*PerClause*] { *Body* }
where *PerClause* is one of
pertarget (*Pointcut*)
perthis (*Pointcut*)
percflow (*Pointcut*)
percflowbelow (*Pointcut*)
issingleton ()

Pointcut definitions *in types*

private pointcut *pc*() : *call(void Foo.m())* ;
a pointcut visible only from the defining type
pointcut *pc*(*int i*) : *set(int Foo.x) && args(i)* ;
a package-visible pointcut that exposes an *int*.
public abstract pointcut *pc*() ;
an abstract pointcut that can be referred to from anywhere.
abstract pointcut *pc*(*Object o*) ;
an abstract pointcut visible from the defining package. Any pointcut that implements this must expose an *Object*.

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

Advice declarations *in aspects*

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

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

Special forms *in advice*

thisJoinPoint
reflective information about the join point.
thisJoinPointStaticPart
the equivalent of **thisJoinPoint.getStaticPart()**, but may use fewer resources.
thisEnclosingJoinPointStaticPart
the static part of the join point enclosing this one.
proceed (*Arguments*)
only available in **around** advice. The *Arguments* must be the same number and type as the parameters of the advice.

Inter-type Member Declarations *in aspects*

int Foo . m (*int i*) { ... }
a method *int m(int)* owned by *Foo*, visible anywhere in the defining package. In the body, **this** refers to the instance of *Foo*, not the aspect.
private *int Foo . m* (*int i*) **throws** *IOException* { ... }
a method *int m(int)* that is declared to throw *IOException,* only visible in the defining aspect. In the body, **this** refers to the instance of *Foo*, not the aspect.
abstract *int Foo . m* (*int i*) ;
an abstract method *int m(int)* owned by *Foo*
Point . **new** (*int x, int y*) { ... }
a constructor owned by *Point*. In the body, **this** refers to the new *Point*, not the aspect.
private static *int Point . x* ;
a static *int* field named *x* owned by *Point* and visible only in the declaring aspect
private *int Point . x = foo()* ;
a non-static field initialized to the result of calling *foo()*. In the initializer, **this** refers to the instance of *Foo*, not the aspect.

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

Other Inter-type Declarations *in aspects*

declare parents : *C* **extends** *D*;
declares that the superclass of *C* is *D*. This is only legal if *D* is declared to extend the original superclass of *C*.

declare parents : C **implements** *I, J* ;
C implements *I* and *J*

declare warning : *set(* Point.*) && !within(Point)* : *"bad set"*;
the compiler warns *"bad set"* if it finds a set to any field of *Point* outside of the code for *Point*

declare error : *call(Singleton.new(..))* : *"bad construction"* ;
the compiler signals an error *"bad construction"* if it finds a call to any constructor of *Singleton*

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

declare precedence : *Security, Logging, ** ;
at each join point, advice from *Security* has precedence over advice from *Logging*, which has precedence over other advice.

general form
- **declare parents** : *TypePat* **extends** *Type* ;
- **declare parents** : *TypePat* **implements** *TypeList* ;
- **declare warning** : *Pointcut* : *String* ;
- **declare error** : *Pointcut* : *String* ;
- **declare soft** : *Type* : *Pointcut* ;
- **declare precedence** : *TypePatList* ;

Primitive Pointcuts

call (*void Foo.m(int)*)
a call to the method *void Foo.m(int)*

call (*Foo.new(..)*)
a call to any constructor of *Foo*

execution (** Foo.*(..) throws IOException*)
the execution of any method of *Foo* that is declared to throw *IOException*

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

initialization (*Foo.new(int)*)
the initialization of any *Foo* object that is started with the constructor *Foo(int)*

preinitialization (*Foo.new(int)*)
the pre-initialization (before the **super** constructor is called) that is started with the constructor *Foo(int)*

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

get (*int Point.x*)
when *int Point.x* is read

set (*!private * Point.**)
when any non-private field of *Point* is assigned

handler (*IOException+*)
when an *IOException* or its subtype is handled with a catch block

adviceexecution()
the execution of all advice bodies

within (*com.bigboxco.**)
any join point where the associated code is defined in the package *com.bigboxco*

withincode (*void Figure.move()*)
any join point where the associated code is defined in the method *void Figure.move()*

withincode (*com.bigboxco.*.new(..)*)
any join point where the associated code is defined in any constructor in the package *com.bigoxco.*

cflow (*call(void Figure.move())*)
any join point in the control flow of each call to *void Figure.move()*. This includes the call itself.

cflowbelow (*call(void Figure.move())*)
any join point below the control flow of each call to *void Figure.move()*. This does not include the call.

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

this (*Point*)
any join point where the currently executing object is an instance of *Point*

target (*java.io.InputPort*)
any join point where the target object is an instance of *java.io.InputPort*

args (*java.io.InputPort, int*)
any join point where there are two arguments, the first an instance of *java.io.InputPort,* and the second an *int*

args (**, int*)
any join point where there are two arguments, the second of which is an *int.*

args (*short, .., short)*
any join point with at least two arguments, the first and last of which are *short*s

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

general form:
- **call**(*MethodPat*)
- **call**(*ConstructorPat*)
- **execution**(*MethodPat*)
- **execution**(*ConstructorPat*)
- **initialization**(*ConstructorPat*)
- **preinitialization**(*ConstructorPat*)
- **staticinitialization**(*TypePat*)
- **get**(*FieldPat*)
- **set**(*FieldPat*)
- **handler**(*TypePat*)
- **adviceexecution()**
- **within**(*TypePat*)
- **withincode**(*MethodPat*)
- **withincode**(*ConstructorPat*)
- **cflow**(*Pointcut*)
- **cflowbelow**(*Pointcut*)
- **if**(*Expression*)
- **this**(*Type* | *Var*)
- **target**(*Type* | *Var*)
- **args**(*Type* | *Var* , …)

where *MethodPat* is:
[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat* (*TypePat* | .., …)
[**throws** *ThrowsPat*]

ConstructorPat is:
[*ModifiersPat*] [*TypePat* .] **new** (*TypePat* | .. , …)
[**throws** *ThrowsPat*]

FieldPat is:
[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat*

TypePat is one of:
- *IdPat* [+] [**[]** …]
- **!** *TypePat*
- *TypePat* **&&** *TypePat*
- *TypePat* || *TypePat*
- (*TypePat*)

Bibliography

[1] Subject-Oriented Programming and Design Patterns, draft, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

[2] Aksit M., Bergmans L. M. J., Vural S., An object-oriented language database integration model: The composition-filters approach. Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP '92), Springer-Verlag Lecture Notes in Computer Science, vol. 615, pp.372-395, June/July 1992.

[3] Aksit M., Bosch J., Sterren W. V. D., Bergmans L. M. J., Real-Time Specification Inheritance Anomalies and Real-Time Filters. Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94), Springer-Verlag Lecture Notes in Computer Science, vol. 821, pp. 386–405, July 1994.

[4] Astels D., Test-Driven Development – a Practical Guide. Prentice Hall PTR 2003.

[5] Austin C., J2SE 5.0 in a Nutshell, Sun Microsystems, May 2004. java.sun.com/developer/technicalArticles/releases/j2se15/

[6] Avgustinov P., Christensen A. S., Hendren L., Kuzins S., Lhoták J., de Moor O., Sereni D., Sittampalam G., Tibble J., abc: An Extensible AspectJ Compiler. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), ACM press, pp. 87-98, Chicago, USA, March 2005.

[7] Batory D., Cardone R., Smaragdakis Y., Object-Oriented Frameworks and Product-Lines. Proceedings of the 1st Software Product Line Conference (SPLC), Kluwer, pp.227-247, Denver, USA, August 2000.

[8] Batory D., Lopez-Herrejon R. E., Martin J.-P., Generating Product-Lines of Product-Families. Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02), pp.81-92, Edinburgh, UK, 2002.

[9] Batory D., O'Malley S., The Design and Implementation of Hierarchical Software Systems With Reusable Components. ACM Transactions of Software Engineering and Methodology, 1(4), pp.355-398, October 1992.

[10] Batory D., Sarvel J. N., Rauschmayer A., Scaling Step-Wise Refinement. Proceedings of the 25th International Conference on Software Engineering (ICSE'03), pp.187-197, Portland, USA, June 2003.

[11] Baumgartner G., Läufer K., Russo V., On the Interaction of Object-Oriented Design Patterns and Programming Languages. Technical Report CSD-TR-96-020, Dept. of Computer Sciences, Purdue University, August 1996.

[12] Beck K., Extreme Programming Explained: Embrace Change. Addison-Wesley 2000.

[13] Beck K., Test Driven Example: By Example. Addison-Wesley, 2003.

[14] Beck K., Gamma E., JUnit: A Cook's Tour. JavaReport 4(5), May 1999. Also available at junit.sourceforge.net/doc/cookstour/cookstour.htm

[15] Beck K., Gamma E., JUnit Cookbook. junit.sourceforge.net/doc/cookbook/cookbook.htm

[16] Beck K., Gamma E., Test Infected: Programmers Love Writing Tests. JavaReport 3(7), July 1998.

[17] Beck K., Johnson R., Patterns Generate Architectures. Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94), Springer-Verlag Lecture Notes in Computer Science, vol. 821, pp. 139-149, Bologna, Italy, July 1994.

[18] Bergmans L. M. J., Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. Ph.D. thesis, University of Twente, Netherlands, 1994.

[19] Bergmans L. M. J., Aksit M., Composing Crosscutting Concerns using Composition Filters. Communications of the ACM, 44(10), pp. 51-57, October 2001.

[20] Bobrow D. G., DeMichiel L. G., Gabriel R. P., Keene S. E., Kiczales G., Moon D. A., Common Lisp Object System Specification X3J13, Document 88-002R. SIGPLAN Notices Special Issue, vol.23, September 1988.

[21] Bosch P., Inheritance vs. delegation: Is one better than the other? Unpublished. www.python.org/ftp/python/doc/delegation.ps.

[22] Bracha G., Cook W., Mixin-Based Inheritance. Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications and European Conference on Object-Oriented Programming (ECOOP/OOPSLA 1990), ACM press, pp. 303-311, Ottawa, Canada, .October 1990.

[23] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996.

[24] Chambers C., Harrison W., Vlissides J., A Debate on Language and Tool Support for Design Patterns. Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2000), ACM press, pp. 277-289, Boston, USA, 2000.

[25] Chiba S., Load-time Structural Reflection in Java. Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000), Springer Verlag Lecture Notes in Computer Science, vol. 1850, pp. 313–336, 2000.

[26] Clarke S.; Composition of Object-Oriented Design Models. Ph.D. thesis, School of Computer Applications Dublin City University, Ireland, January 2001.

[27] Clarke S.; Harrison W., Ossher H., Tarr P. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. Proceedings of the 14th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 1999), ACM press, pp. 325-339, Denver, USA, 1999.

[28] Clement A., Colyer A., Kersten M., Aspect-Oriented Programming with AJDT. Workshop on Analysis of Aspect-Oriented Software at ECOOP 2003, Darmstadt, Germany, July 2003.

[29] Cole L., Borba P., Deriving Refactorings for AspectJ. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), ACM press, pp. 123-134, Chicago, USA, March 2005.

[30] Cole L., Borba P., Using Programming Laws to Modularize Concurrency in a Replicated Database Application, 1st Brasilian Workshop on Aspect-Oriented Software Development – WBSOA'04 – SBES'04, Brazil, October 2004.

[31] Colyer A., Clement A., Large-scale AOSD for Middleware. Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), ACM press, pp. 56-65, Lancaster, UK, March 2004.

[32] Cooper J., Java Design Patterns: A Tutorial, Addison-Wesley 2000.

[33] Cox B.J., Object Oriented Programming: An Evolutionary Approach, Addison Wesley, 1987.

[34] Czarnecki K.; Eisenecker U. W., Generative Programming - Methods, Tools, and Applications, Addison Wesley, 2000.

[35] Czarnecki K., Eisenecker U., Glück R., Vandevoorde D., Veldhuizen T., Generative Programming and Active libraries. Proceedings of the Dagstuhl Seminar 98171 in Generic Programming, Schloss Dagstuhl, Germany, April 1998. Published Springer-Verlag Lecture Notes of Computer Science, vol.1766 , pp.25-39, 1999.

[36] van Deursen A., Marin M., Moonen L., Aspect Mining and Refactoring. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03), Waterloo, Canada, November 2003.

[37] Dijsktra E., A Discipline of Programming, Prentice Hall, 1976.

[38] Dijkstra E., Go-to Statement Considered Harmful, Communications of the ACM, 11 (3), March 1968.

[39] Eckel B., Thinking in Patterns, revision 0.9. Book in progress, May 20, 2003. Available at 64.78.49.204/TIPatterns-0.9.zip

[40] Elrad T. (moderator) with panelists Aksit M.; Kiczales G., Lieberherr K., Ossher H., Discussing Aspects of AOP. Communications of the ACM, pp. 33-38, October 2001.

[41] Ettinger R., Verbaere M., Untangling: A Slice Extraction Refactoring. Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), ACM press, pp. 93-101, Lancaster, UK, March 2004.

[42] Fayad M., Schmidt D., Johnson R., Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley & Sons, 1999.

[43] Fernandes S., Cachopo J., Silva A. R., Supporting Evolution in Workflow Definition Languages. Proceedings of the 20th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'2004), Springer-Verlag Lecture Notes of Computer Science, vol.2932, pp. 208-217, Merin, Czech Republic, January 2004.

[44] Filman R. E., Elrad T. H., Clarke S., Aksit M., Aspect-Oriented Software Development. Addison-Wesley 2005.

[45] Filman R. E., Friedman D. P., Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns at OOPSLA 2000, Minneapolis, October 2000.

[46] Foote B., Johnson R. E., Reflective Facilities in Smalltalk-80. Proceedings of the 4th Conference on Object-Oriented Programming Systems, Languages and Applications, (OOPSLA'89), New Orleans, USA. ACM press, pp. 327-335, October 1989.

[47] Fowler M. (with contributions by Beck K., Opdyke W., Roberts D.). Refactoring – Improving the Design of Existing Code. Addison Wesley 1999.

[48] Gamma E.; Helm R., Johnson R., Vlissides J., Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[49] Garcia A., Sant'Anna C., Figueiredo E., Kulesza U., Lucena C., Staa A., Modularizing Design Patterns with Aspects: A Quantitative Study. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), ACM press, pp. 3-14, Chicago, USA, March 2005.

[50] Goldberg A., Robson D., Smalltalk 80: The Language and its Implementation. Addison-Wesley, 1983.

[51] Gosling J., Joy B., Steele G., The Java Language Specification, second edition. Addison Wesley, 1996.

[52] Grand M., Patterns in Java, volume 1. John Wiley & Sons, 1998.

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

[54] Griswold W. G., Notkin D., Automated assistance for program restructuring. ACM Transactions on Software Engineering and Methodology, 2(3), pp. 228–269, July 1993.

[55] Gudmundson S., Kiczales G., Addressing practical software development issues in AspectJ with a pointcut interface. Workshop on Advanced Separation of Concerns at ECOOP 2001, June 2001.

[56] Gurp J. v., Bosch J., Design Erosion: Problems & Causes. Journal of Systems & Software, 61(2), pp. 105-119, Elsevier Science, pp. 105-119, March 2002.

[57] Hanenberg S., Hirschfeld R., Unland R., Kawamura K., Applying Aspect-Oriented Composition to Framework Development - A Case Study, First International Workshop on Foundations of Unanticipated Software Evolution (FUSE 2004), Barcelona, Spain, March 28, 2004.

[58] Hanenberg S., Oberschulte C., Unland R., Refactoring of Aspect-Oriented Software, Net.ObjectDays 2003, Erfurt, Germany, September 2003.

[59] Hannemann J., Fritz T., Murphy G., Refactoring to Aspects – an Interactive Approach. Eclipse Technology eXchange (ETX) Workshop at OOPSLA 2003, October 2003.

[60] Hannemann J., Kiczales G., Design Pattern Implementation in Java and AspectJ. Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), Seattle, USA, ACM press, pp. 161-173, November 2002.

[61] Hannemann J., Murphy G., Kiczales G., Role-Based Refactoring of Crosscutting Concerns. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, USA, ACM press, pp. 135-146, March 2005.

[62] Harbulot B., An Investigation of Aspect-Oriented Programming, MSc thesis, Faculty of Science and Engineering, University of Manchester, UK, October 2002.

[63] Harrison W., Ossher H., Subject-oriented programming (a critique of pure objects). Proceedings of the 8th annual Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93), Washington, D.C., USA, ACM press, pp. 411-428, September 1993.

[64] Hillsdale E., Hugunin J. Advice Weaving in AspectJ. Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2003), Lancaster, UK, ACM press, pp. 26-35, March 2004.

[65] Hollingsworth D., The Workflow Reference Model. Document Number TC-00-1003. Issue 1.1. 19 January 1995.

[66] Holmes D., Synchronisation Rings – Composable Synchronisation for Object-Oriented Systems. Ph.D. thesis, Macquarie University, Sydney, Australia, October 1999.

[67] Jacobson I., Christerson M., Jonsson P., Övergaard G., Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley 1992.

[68] Johnson R. E., Frameworks = (Components + Patterns). Communications of the ACM, 40(10), 39-42, 1997.

[69] Johnson R. E., Foote B., Designing Reusable Classes. Journal of Object-Oriented Programming, pp.22-35, June/July 1988.

[70] Johnson R. E., Opdyke W. F., Refactoring and Aggregation. Object Technologies for Advanced Software, First JSSST International Symposium, Springer-Verlag Lecture Notes in Computer Science, vol.742, pp.264–278, November 1993.

[71] Johnson R. E., Woolf B., Type Object. Chapter 4 of Pattern Languages of Program Design 3 (Martin R., Riehle D., Buschmann F., editors), Software Patterns Series, Addison-Wesley, October 1997.

[72] Joyner I., "C++?? A Critique of C++ and Programming and Language Trends of the 1990s", 3rd Edition. Unpublished book. Available at atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/tools/java/misc/ACritiqueOfC++.pdf

[73] Iwamoto M., Zhao J., Refactoring Aspect-Oriented Programs, 4th AOSD Modeling With UML Workshop at UML'2003, San Francisco, USA, October 2003.

[74] Kang K. C., Cohen S. G., Hess J. A., Novak W. E., Peterson A., Feature-Oriented Domain Analysis Feasibility Study, SEI. Technical Report CMU/SEI-90-TR-21, November 1990.

[75] Keene S. E., Object-Oriented Programming in Common Lisp: A programmer's guide to CLOS, Addison-Wesley, 1989.

[76] Kerievsky J., Refactoring to Patterns, Addison-Wesley, 2004.

[77] Kersten M., AOP tools comparison, parts 1 and 2. DeveloperWorks (first article of the AOP@Work series), IBM, February 2005.

[78] Kersten M., Murphy G., Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-oriented Programming. Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'99), Denver, USA, ACM press, pp. 340-352, November 1999.

[79] Khoshafian S., Copeland G., Object Identity. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA 1986), Portland, USA, ACM press, pp. 406-416, 1986. Also appeared in Readings in Object-Oriented Database Systems (Zodnik S., Maier D. editors), San Mateo, CA: Morgan Kaufman Publishers, 1990.

[80] Khoshafian S., Abnous R., Object Orientation, second edition, John Wiley & Sons, 1995.

[81] Kiczales G., Towards Open Implementations – a New Model of Abstraction in the Engineering of Software, proceedings of the workshop on Reflection and Meta-level Architectures at IMSA'92, Tokyo, Japan, November 1992.

[82] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., An Overview of AspectJ. Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), Budapest, Hungary, Springer Verlag Lecture Notes in Computer Science, vol. 2072, pp. 327-353, June 2001.

[83] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., Getting Started with AspectJ. Communications of the ACM, 44(10):59-65, October 2001.

[84] Kiczales G., Hugunin J., Kersten M., Lamping J., Lopes C., Griswold W. G., Semantics-Based Crosscutting in AspectJ. Proceedings of the workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE'2000, Limerick, Ireland, June 2000.

[85] Kiczales G., Lamping J., Lopes C., Maeda C., Mendhekar A., Open Implementation Design Guidelines. Proceedings of the International Conference on Software Engineering (ICSE'97), Boston, USA, ACM press, pp. 481-490, May 1997.

[86] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J., Aspect-Oriented Programming. Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland, Springer-Verlag Lecture Notes in Computer Science, vol. 1241, pp. 220-242, June 1997.

[87] Kiczales G., Rivières J. des, Bobrow D., The Art of the Metaobject Protocol. MIT Press, 1991.

[88] Kölling M., The Design of an Object-Oriented Environment and Language for Teaching, Ph.D. thesis, Basser Department of Computer Science, University of Sydney, Australia, 1999.

[89] Koppen C., Störzer M., PCDiff: Attacking the Fragile Pointcut Problem, Interactive Workshop on Aspects in Software (EIWAS) 2004, Berlin, Germany, September 2004.

[90] Laddad R., AspectJ in Action – Practical Aspect-Oriented Programming, Manning 2003.

[91] Laddad R., Aspect-Oriented Refactoring, parts 1 and 2, The Server Side, 2003. www.theserverside.com/

[92] Lieberherr K. J., Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996.

[93] Lieberherr K., Holland I., Assuring Good Style for Object-Oriented Programs, IEEE Software, pp. 38-48, September 1989.

[94] Lieberherr K., Holland I., Formulations and Benefits of the Law of Demeter, ACM press, SIGPLAN Notices 24(3), pp.67-78, March 1989.

[95] Lieberherr K., Holland I., Object-Oriented Programming: An Objective Sense of Style. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA 88), ACM SIGPLAN Notices, 23 (11), November 1988.

[96] Lieberherr K., Orleans D., Ovlinger J., Aspect-Oriented Programming with Adaptive Methods. Communications of the ACM, 44(10), p.39-41, October 2001.

[97] Lieberman H., Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'86), Portland, USA, ACM press, pp. 214-223, September 29-October 2 1986.

[98] Lippert M., Lopes C. V., A Study on Exception Detection and Handling Using Aspect-Oriented Programming, Technical Report P9910229 CSL-99-1, Xerox Palo Alto Research Center, December 1999.

[99] Liskov B., Data Abstraction and Hierarchy, Keynote Address. Addendum to the Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'87), pp. 17-34, October 1987. Printed as SIGPLAN Notices 23(5) , May 1988.

[100] Lopes C. V., D: A Language Framework for Distributed Computing, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, USA, November 1997.

[101] Lopes C. V., Dourisch P., Lorenz D., Lieberherr K., Beyond AOP: Toward Naturalistic Programming. Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA 2003), ACM press, pp. 198-207, October 2003.

[102] Lopes C. V., Kiczales G., D: A Language Framework for Distributed Programming, Xerox PARC, Palo Alto, CA. Technical report SPL97-010 P9710047, February 1997.

[103] Maes P., Concepts and Experiments in Computational Reflection. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'87), Orlando, USA, ACM press, pp. 147-155, December 1987.

[104] Manolescu D-A., A Micro Workflow Architecture Supporting Compositional Object-Oriented Software Development, Ph.D. thesis, University of Illinois at Urbana-Champaign, USA, 2001.

[105] Manolescu D.-A., Understanding Large Systems, Software Archeology workshop at OOPSLA 2001, Tampa Bay, Florida, October 2001.

[106] Matsuoka S., Yonezawa A., Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. Research Directions in Concurrent Object-Oriented Programming (Agha G., Wegner P., et al., editors), pp. 107-150, MIT press, 1993.

[107] McEachen N., Alexander R. T., Distributing Classes with Woven Concerns – An Exploration of Potential Fault Scenarios. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, USA, ACM press, pp. 192-200, March 2005.

[108] McHale C., Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance. Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, Ireland, October 1994.

[109] Mendhekar A., Kiczales G., Lamping J., RG: A Case-Study for Aspect-Oriented Programming, Technical Report SPL97-009 P9710044, Xerox Palo Alto Research Center, February 1997.

[110] Meyer B., Design by Contract, Advances in Object-Oriented Software Engineering, Prentice Hall, 1992.

[111] Meyer B., Eiffel: The Language. Prentice Hall, 1991.

[112] Meyer B., Genericity versus Inheritance, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'86), ACM press, pp. 391-405, September 1986.

[113] Meyer B., Object-Oriented Software Construction, second edition, Prentice Hall, 1997.

[114] Mikhajlov L., Sekerinski E., A Study of The Fragile Base Class Problem. Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP 1998), Brussels, Belgium, Springer-Verlag Lecture Notes In Computer Science, vol. 1445, pp. 355-382, July 1998.

[115] Monteiro M. P., Catalogue of Refactorings for AspectJ, Technical Report UM-DI-GECSD-200402, Departmento de Informática, Universidade do Minho, December 2004. Available at www.di.uminho.pt/~jmf/PUBLI/papers/ 2004-TR-02.pdf

[116] Monteiro M. P., Refactoring Legacy Objects to Aspects, poster presented at the student extravaganza session at the AOSD 2003, Boston, USA, March 2003. Available at gec.di.uminho.pt/mpm/Poster%20AOSD2003.pdf

[117] Monteiro M. P., Fernandes J. M., C++ é Inadequado para Ensinar OO. Ingenium, Ordem dos Engenheiros, 2ª série, nr.69, pp.76-8, September 2002.

[118] Monteiro M. P., Fernandes J. M., Object-to-Aspect Refactorings for Feature Extraction, Industry track paper at the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, UK, March 2004.

[119] Monteiro M. P., Fernandes J. M., Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. Proceedings of the DSOA'2004 workshop at JISBD 2004 (IX Jornadas de Ingeniería de Software y Bases de Datos), Málaga, Spain, November 2004.

[120] Monteiro M. P., Fernandes J. M., Refactoring a Java Code Base to AspectJ – An Ilustrative Example, submitted to the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary, September 2005.

[121] Monteiro M. P., Fernandes J. M., Some Thoughts On Refactoring Objects to Aspects. Proceedings of the DSOA'2003 workshop at JISBD 2003 (VIII Jornadas de Ingeniería de Software y Bases de Datos), Alicante, Spain, November 2003.

[122] Monteiro M. P., Fernandes J. M., The Search for Aspect-Oriented Refactorings Must Go On. Position paper for the LATE workshop at AOSD'05, Chicago, USA, March 2005.

[123] Monteiro M. P., Fernandes J. M., Towards a Catalogue of Aspect-Oriented Refactorings. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, USA, ACM press, pp. 111-122, March 2005.

[124] Nierstrasz O., Tsichritzis D. (editors), Object-Oriented Software Composition, Prentice Hall, 1995.

[125] Opdyke W. F., Refactoring Object-Oriented Frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign, USA, 1992.

[126] Opdyke W. F., Johnson R. E., Creating Abstract Superclasses by Refactoring. Proceedings of the 1993 ACM Conference on Computer Science, ACM press, pp.66-73, 1993.

[127] Opdyke W. F., Johnson R. E., Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA '90), September 1990.

[128] Orleans D., Separating behavioral concerns with predicate dispatch, or, if statement considered harmful, workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA 2001, Tampa Bay, USA, October 2001.

[129] Ossher H., Tarr P., Multi-dimensional separation of concerns and the Hyperspace approach. Software Architectures and Component Technology (Aksit M., editor), Kluwer, pp.293-323, 2002.

[130] Ossher H., Tarr P., The Shape of Things to Come: Using Multi-Dimensional Separation of Concerns with Hyper/J to (Re)Shape Evolving Software, Research Report RC 20946, IBM Thomas J. Watson Research Center, July 2001.

[131] Ossher H., Tarr P., Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. Communications of the ACM 44(10), pp. 43-50, October 2001.

[132] Parnas D. L., On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12), pp. 1053-1059, December 1972.

[133] Parnas D. L., Software Aging. Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), Sorrento, Italy, ACM press, pp. 279-287, May 1994.

[134] Popovici A., Alonso G., Gross T., Just-In-Time Aspects: Efficient Dynamic Weaving for Java. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), Boston, USA, ACM press, pp. 100-109, March 2003.

[135] Pree W., Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

[136] Roberts D. B., Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign, USA, 1999.

[137] Roberts D. B., Brant J., Johnson R. E., A refactoring tool for smalltalk. Theory and Practice of Object Systems 3(4), pp. 253–263, 1997.

[138] Robillard M., Murphy G., Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), ACM press, pp. 406-416, May 2002.

[139] Rura S., Refactoring Aspect-Oriented Software. Bachelor thesis, Williams College, Williamstown, Massachusetts, USA, May 2003.

[140] Sabbah D., Aspects – from Promise to Reality. Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, UK, ACM press, pp. 1-2, March 2004.

[141] Sakkinen M., Comments on "the Law of Demeter" and C++. ACM SIGPLAN Notices 23(12), pp.38-44, December 1988.

[142] Shepherd D., Pollock L., Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report No. 2004-03. Dept. of Computer & Information Sciences, University of Delaware, Newark, DE, 2003.

[143] Schmidt D., Stal M., Rohnert H., Buschmann F., Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley & Sons, 1999.

[144] Silva A. R., Programação Concorrente com Objectos: Separação e Composição de Facetas com Padrões de Desenho, Linguagem de Padrões e Moldura de Objectos,

Ph.D. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal, March 1999.

[145] Snyder A., Encapsulation and Inheritance in Object-Oriented Programming Languages. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'86), Portland, USA, ACM press, pp. 38-45, September 1986.

[146] Soares S., Laureano E., Borba P., Implementing Distribution and Persistence Aspects with AspectJ. Proceedings da 17th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002), Seattle, EUA, ACM press, pp. 174-190, November 2002.

[147] Stroustrup B., The C++ Programming Language, third edition. Addison-Wesley 1997.

[148] Stroustrup B., The Design and Evolution of C++. Addison Wesley 1994.

[149] Szyperski C., Component Software – Beyond Object-Oriented Programming, Addison-Wesley/ACM press, 1998.

[150] Tarr P., Ossher H., Harrison W., Sutton Jr., S.M., N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, USA, IEEE Computer Society press, pp. 107-119, May 1999.

[151] Tokuda L., Batory D., Automating three modes of evolution for object-oriented software architecture. Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99), San Diego, USA, May 1999.

[152] Tokuda L., Batory D., Evolving object-oriented designs with refactorings. Proceedings of 14th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, p.174, October 1999.

[153] Tonella P., Ceccato M., Migrating Interface Implementation to Aspects. Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, USA, IEEE Computer Society, pp. 220-229, September 2004.

[154] Tourwé T., Brichau J., Gybels K., On the Existence of the AOSD-Evolution Paradox, Workshop on Software-engineering Properties of Languages for Aspect Technologies at AOSD 2003, Boston, USA, March 2003.

[155] Udell J., Componentware. Byte, 19(5), pp. 46–56, May 1994.

[156] Vermeulen A., Ambler S. W., Bumgardner G., Metz E., Misfeldt T., Shur J., Thompson P., The Elements of Java Style. Cambridge University Press, 2000.

[157] Vlissides J., Pattern Hatching – Design Patterns Applied. Addison-Wesley 1998.

[158] Wake W., Refactoring Workbook, Addison Wesley, 2004.

[159] Walker R. J., Baniassad E. L. A., Murphy G. C., An Initial Assessment of Aspect-Oriented Programming. Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, USA, IEEE Computer Society press, pp. 120-130, May 1999.

[160] Wegner P., Dimensions of Object-Based Language Design. Proceedings of the OOPSLA'87, ACM press, Sigplan Notices 23(11), pp. 168-182, October 1987.

[161] Whitehead K., Component-based Development – Principles and Planning for Business Systems. Addison Wesley Component Software Series (C. Szyperski, series Editor), 2002.

[162] Wloka J., Refactoring in the Presence of Aspects, ECOOP2003 PhD workshop, July 2003.

[163] Woolf B., Null Object, Chapter 1 of Pattern Languages of Program Design 3 (Martin R., Riehle D., Buschmann F., editors), Software Patterns Series, Addison-Wesley, October 1997.

[164] Zhang C., Jacobsen H.-A., A Prism for Research in Software Modularization Through Aspect Mining. Technical Communication, Middleware Systems Research Group, University of Toronto, September 2003.

[165] Zhang C., Jacobsen H.-A., Quantifying Aspects in Middleware Platforms. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), Boston, USA, ACM press, pp. 130-139, March 2003.

[166] Zhang C., Jacobsen H.-A., Refactoring Middleware with Aspects. IEEE Transactions on Parallel and Distributed Systems 14(11), pp. 1058-1073, November 2003.

[167] Zhao J., Towards a Metrics Suite for Aspect-Oriented Software, Technical-Report SE-2002-136-25, Information Processing Society of Japan (IPSJ), 2002.

References from the Internet

[168] AHEAD tool suite. www.cs.utexas.edu/users/schwartz/ATS.html

[169] AJDT home page. www.eclipse.org/ajdt

[170] Aspect-Oriented Software Development home page. aosd.net/

[171] AspectJ and AspectWerkz teams, AspectJ and AspectWerkz to Join Forces, joint announcement, January 19th 2005. dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/aj5announce.html and aspectwerkz.codehaus.org/index-merge.html

[172] AspectJ home page, http://www.eclipse.org/aspectj/

[173] AspectJ Team, The AspectJ 5 Development Kit Developer's Notebook, 2004. dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/ajdk15notebook/index.html

[174] AspectJ users mailing list, https://dev.eclipse.org/mailman/listinfo/aspectj-users

[175] AspectWerkz home page, aspectwerkz.codehaus.org/

[176] Composition Filters homepage trese.cs.utwente.nl/composition_filters/

[177] Design Pattern Java Companion home page, www.patterndepot.com/put/8/JavaPatterns.htm

[178] Design Pattern home page of Vince Huston, home.earthlink.net/~huston2/dp/patterns.html

[179] eclipse home page. www.eclipse.org/

[180] FluffyCat page for Design Patterns in Java, www.fluffycat.com/java/patterns.html

[181] GoF's Design Patterns in Java by Franco Guidi Polanco www.cim.polito.it/people/guidi/DesignPatternsJava.htm

[182] JHotDraw home page www.jhotdraw.org

[183] JUnit home page. www.junit.org/

[184] Refactoring home page, www.refactoring.com/

[185] Refactoring mailing list at Yahoo, groups.yahoo.com/group/refactoring/

[186] Software Engineering Group at INESC-ID home page, www.esw.inesc.pt/

[187] WorkSCo project home page. www.esw.inesc-id.pt/worksco/