UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

EDUARDO KESSLER PIVETA

Improving the Search for Refactoring Opportunities on Object-Oriented and Aspect-Oriented Software

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. D. Phil. Roberto Tom Price
Advisor

Prof. Dr. Marcelo Soares Pimenta
Coadvisor

Porto Alegre, January 2009

CATALOGAÇÃO NA PUBLICAÇÃO

Piveta, Eduardo Kessler

Improving the Search for Refactoring Opportunities on Object-Oriented and Aspect-Oriented Software / Eduardo Kessler Piveta. – Porto Alegre: Programa de Pós Graduação em Computação, 2009.

235 f.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Doutorado em Ciência da Computação, Porto Alegre, BR–RS, 2009. Advisor: Roberto Tom Price; Coadvisor: Marcelo Soares Pimenta.

1. Refatoração. 2. Desenvolvimento de Software Orientado a Aspectos. 3. Evolução de Software. I. Price, Roberto Tom. II. Pimenta, Marcelo Soares. III. Título.

ACKNOWLEDGEMENTS

I would like to thank all the people who contributed to the development of this work.

First, I would like to thank Tom Price and Marcelo Pimenta, for their ideas, suggestions, comments and discussions. Their help was fundamental to the conclusion of this thesis.

Also, I would like to thank Ana Moreira, João Araújo and Pedro Guerreiro, for their excellent hospitality and for their immense help on my work during my stay in Portugal.

Special thanks go to my family: Gilda, Enio and Juliana, who supported me unconditionally during these long four years, and to my beloved Deise, who shared the good and bad moments of this journey, helped me and motivated me when I needed most.

Also, my thanks to all the professors who participated directly or indirectly in the development of this work: Maria Lúcia Lisboa, Álvaro Moreira, Carlos Eduardo Pereira, Carla Freitas, Leandro Wives, Daltro Nunes, Paulo Borba, Paulo Masiero, Toacy Cavalcante, Sérgio Soares, and Marco Túlio Valente. And my thanks to the colleagues of UFRGS for their friendship and for the coffee-driven relaxing moments at Campus do Vale, and to the colleagues at UNL for the guinness-driven tours at Portugal.

I also acknowledge the funding support by CAPES and CNPq, which financially helped me in my stay in Porto Alegre and Costa da Caparica.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

AHP	Analytic Hierarchy Process
AJDT	AspectJ Development Tools
AO	Aspect-Oriented
AOSD	Aspect-Oriented Software Development
AST	Abstract Syntax Tree
CAE	Coupling on Advice Execution
CDA	Crosscutting Degree of an Aspect
CBSD	Component-Based Software Development
DIT	Depth of Inheritance Tree
GQM	Goal-Question Metric
GoF	Gang-of-Four
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
JMX	Java Management Extensions
JSP	Java Server Pages
J2EE	Java 2 Enterprise Edition
LOCC	Lines of Code
MCDM	Multi-Criteria Decision Method
DFA	Deterministic Finite Automata
NOC	Number of Children
NOM	Number of Operations in Module
OO	Object-Oriented
RO	Refactoring Opportunity
RP	Refactoring Pattern
SQA	Software Quality Assurance
UML	Unified Modelling Language

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Refactoring is the process of improving the design of software systems without changing their externally observable behaviour. Refactoring can help to incrementally improve the quality of a software system through the application of behavioural preserving transformations called refactoring patterns.

The main goal of the research this thesis reports is to provide a detailed process for refactoring, including mechanism for (i) the selection and creation of quality models, the selection of refactoring patterns, and the creation and use of heuristic rules, (ii) the search for refactoring opportunities and prioritisation, (iii) the assessment of the effects of refactoring on software quality, and (iv) the trade-off analysis and the application of refactoring patterns.

To exemplify how the approach works and how the process can be used, different case studies are being used throughout the thesis. The selection of refactoring patterns, quality models and heuristic rules, and the search and prioritisation mechanisms are exemplified for object-oriented software. The assessment of refactoring effects on software quality, the trade-off analysis and the application of refactoring patterns are discussed in the context of aspect-oriented programming.

Besides the definition of a refactoring process, a set of additional contributions of this thesis are (i) the definition of an approach for the selection and ranking of refactoring patterns, (ii) an approach for reducing the search space for refactoring opportunities when dealing with successive refactoring, (iii) the definition of search mechanisms for refactoring opportunities, (iv) an approach to evaluate the effects of refactoring on software quality, (v) a catalogue of shortcomings in aspect-oriented software and their related refactoring patterns, and (vi) an study of metrics for aspect-oriented software, including their formal definition, analytical evaluation and data interpretation.

Keywords: Refatoração, Desenvolvimento de Software Orientado a Aspectos, Evolução de Software Refactoring, Aspect-Oriented Software Development, Software Evolution

Melhorando a Busca por Oportunidades de Refatoração em Software Orientado a Objetos e Orientado a Aspectos

RESUMO

Refatoração é o processo de melhorar o projeto de sistemas de software sem modificar seu comportamento externamente observável. O processo de refatoração pode auxiliar a incrementalmente melhorar a qualidade de software de um sistema através da aplicação de transformações que preservam comportamento chamadas de padrões de refatoração.

O principal objetivo da pesquisa que esta tese descreve é prover um processo detalhado para refatoração, incluindo mecanismos para (i) seleção e criação de modelos de qualidade, padrões de refatoração e funções heurísticas, (ii) a busca e priorização de oportunidades de refatoração, (iii) a avaliação dos efeitos da refatoração na qualidade de software e (iv) a análise de vantagens e desvantagens e a aplicação de padrões de refatoração.

Para exemplificar como os mecanismos propostos funcionam e como o processo pode ser usado, diferentes estudos de caso são usados ao longo da tese. A seleção dos padrões de refatoração, dos modelos de qualidade e das funções heurísticas são exemplificados para software orientado a objetos, bem como os mecanismos de busca e priorização. A avaliação dos efeitos da refatoração na qualidade de software, a análise de vantagens e desvantagens e a aplicação de padrões de refatoração são discutidos no contexto de programação orientada a aspectos.

Além da definição de um processo de refatoração, um conjunto adicional de contribuições desta tese são: (i) a definição de uma abordagem para a seleção e ranking de padrões de refatoração baseada no AHP (*Analytic Hierarchy Process*), (ii) uma abordagem para reduzir o espaço de busca para oportunidades de refatoração ao manipular refatorações sucessivas, (iii) a definição de mecanismos de busca para oportunidades de refatoração, (iv) uma abordagem para avaliar os efeitos de refatoração na qualidade de software, (v) um catálogo de problemas encontrados em software orientado a aspectos e seus padrões de refatoração associados, e também um conjunto de recomendações para evitar estes problemas e (vi) um estudo de métricas orientadas a aspectos, incluindo a sua definição formal, avaliação analítica e interpretação de dados.

Palavras-chave: Refatoração, Desenvolvimento de Software Orientado a Aspectos, Evolução de Software.

1 INTRODUCTION

Refactoring (OPDYKE, 1992; FOWLER et al., 1999) is the process of improving the design of software systems without changing their externally observable behaviour. Refactoring can help to incrementally improve the software quality of a software system through the application of behavioural preserving transformations called refactoring patterns. A refactoring pattern is comprised by a name, a set of parameters, a motivation, a set of mechanics (including pre and postconditions) and an example.

The refactoring process is structured to cope with the following activities (MENS; TOURWE, 2004): (i) identifying where to apply refactoring patterns, (ii) assessing the effects of refactoring on quality, (iii) maintaining consistency of the pieces of software being restructured by refactoring, and (iv) guaranteeing that the application of refactoring patterns preserve software behaviour. This thesis focuses on the first two activities.

The *identification of where to apply refactoring patterns* is the search for pieces of software for which a refactoring pattern can be applied. This identification considers deficiencies, inadequacies or incompleteness that pieces of software can have, called *shortcomings*. In this context, a refactoring opportunity is defined as the association between a piece of software, a shortcoming, and a refactoring pattern. The problem is that the number of refactoring opportunities can be high (FOWLER et al., 1999). It is interesting, then, that the developer can narrow the search for refactoring opportunities to a sub-set of refactoring patterns, shortcomings and pieces of software.

Additionally, the application of a single refactoring pattern may not be enough to bring substantial benefits to the quality of a piece of software. Current research projects (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006; TOURWE; MENS, 2003) focus on improvements considering the application of a single refactoring pattern and not on the application of sequences of refactoring patterns. Also, the number of refactoring sequences can be high, as the number of sequences is given by the combination of refactoring patterns. There is the need for an approach to reduce the number of refactoring sequences to evaluate.

The *assessment of the effects of refactoring on quality* typically focuses on quantitative evaluations, using functions called impact functions. An impact function is a mathematical function which computes the expected value of a metric in case a specific refactoring pattern is applied, without actually applying the refactoring pattern (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006). Unfortunately, an approach for the creation of impact functions does not exist, and there are no defined impact functions for other paradigms besides object-orientation (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006).

The *consistency maintenance* of the artefacts being restructured by refactoring is another research area regarding refactoring. For example, whenever a source code class

changes, there is the need to update the related design models. Or, if a class diagram is changed, the sequence diagrams referencing the classes in the diagram must also be changed accordingly.

Another related activity is to *guarantee that the refactoring patterns preserve software behaviour*. Several formalisms have been used to help in this activity, including the use of assertions (preconditions, postconditions and invariants) and graph transformations. Cornelio (CORNELIO, 2004) provides an extensive discussion of software behaviour preservation and refactoring.

These activities (consistency maintenance and guaranteeing behavioural preservation) are outside of the scope of this thesis.

The open issues addressed by this thesis are related to the following goals: (a) identification of refactoring opportunities and (b) evaluation of the effects of refactoring on the software quality. These goals can be better meet as a set of activities which are part of a software development process, which means organizing the activities and distributing responsibilities for the roles of the process (or for new roles).

The activities regarding the identification of refactoring opportunities and the assessment of the effects of refactoring on software quality must be generic enough to be applicable to different phases of software development, including analysis, design, implementation and testing. The refactoring process can be structured as part of a planned software development process (such as RUP (KRUTCHEN, 2000)), with defined activities, roles and artefacts or expressed as a set of practices and guidelines to be included in the practices and guidelines of an agile process (COCKBURN; HIGHSMITH, 2001), such as XP (BECK, 1999) or Scrum (SCHWABER, 1995).

In summary, this thesis focuses on the improvement of software quality through refactoring. The main goal is to provide a set of detailed activities and techniques for: (i) searching and prioritising refactoring opportunities, (ii) ranking refactoring patterns according to a quality model, (iii) evaluating the effects of refactoring on software quality and (iv) analysing the trade-offs of applying refactoring patterns. Different case studies are being used throughout the thesis to exemplify how the proposed activities and techniques could be implemented.

The following sections are organised as follows. Section 1.1 describes the main motivations for this work. Section 1.2 highlights the contributions while Section 1.3 shows how each contribution is evaluated. Section 1.4 describes how the thesis is organised.

1.1 Motivation

Different systems can have different needs in terms of the expected software quality. For example, in embedded systems memory is an important characteristic, while in real-time systems timing is more important. For other kinds of systems, such as information systems, the main quality characteristics could be reusability, usability and portability, for example.

In software engineering, a *quality attribute* is a requirement which specifies criteria that can be used to judge the operation of a system (ISO, 2001). Other terms for quality attributes are non-functional requirements, constraints, quality goals and quality of service requirements. Common examples of quality attributes, according to Boehm (BOEHM; IN, 1996), are: assurance, interoperability, performance, evolvability, cost and reusability. The ISO/IEC-9126 (ISO, 2001), the international standard for product quality and quality models, describes several quality attributes for software systems. Quality attributes are

usually described in *quality attributes catalogues*, which are organised collections that provide details of how each quality attribute can affect the properties of software, how each quality attribute relates to other quality attributes, and of possible trade-offs between conflicting quality attributes (BOEHM; IN, 1996; ISO, 2001; MYLOPOULOS; CHUNG; NIXON, 1992).

These quality attributes can be grouped together, organised hierarchically and refined to express the expected quality of a specific software system or of a specific domain. Such groups of related quality attributes are called *quality models*. Several quality models have been proposed for specific domains, such as: component-based software development (CBSD) (BERTOA; VALLECILLO, 2002), quality attributes that cross-cut requirements at an early stage of the software development process (MOREIRA; ARAUJO; BRITO, 2002), quality attributes for web software (OFFUTT, 2002) and quality attributes for the specification of software architectures (KAZMAN; BASS, 1994).

Refactoring activities usually aim at improving these quality attributes, including improvements on the software artefacts, making them easier to maintain, to comprehend and to change. In agile methodologies, these activities are an integral part of the software development process. These methodologies use test cases to drive the creation of software applications and to ensure that the application of refactoring patterns does not change the behaviour of the application.

The first issue addressed in this thesis is how to order a set of refactoring patterns according to their positive contribution to the expected quality attributes of a given piece of software. This ordering produces a ranking of refactoring patterns in terms of a quality model. Currently, there are no automated mechanisms to rank refactoring patterns in terms of a quality model. The use of an approach for ranking refactoring patterns in terms of quality attributes can be used to optimise the search for refactoring opportunities, enabling the developer to focus on the refactoring patterns that improve the quality attributes most.

This thesis is meant to be general enough to encompass refactoring activities in different phases of software development. Therefore, the term *software element* is used to denote pieces of software ranging from source code to analysis and design models. For example, in the context of object-oriented source code, software elements can be packages, classes, methods, attributes. In case of aspect-oriented software, software elements are aspects, advices, pointcuts, and inter-type declarations. In use case diagrams, they are the actors, use cases, and relationships.

Once the refactoring patterns are selected, the second issue is to find where the refactoring patterns can be applied on a piece of software (i.e. identify the refactoring opportunities). This includes the selection of the pieces of software to be analysed, the metrics to be computed, and the generation of a list of refactoring opportunities.

The third issue deals with the quantitative evaluation of the effects of refactoring on software quality. During refactoring activities, the developer has to correctly evaluate the trade-offs between refactoring patterns in terms of the affected quality attributes. As these quality attributes can be conflicting with each other (BOEHM; IN, 1996), the task of selecting optimal or near-optimal applications of the selected refactoring patterns can be hard. Additionally, it is not always clear to the developer how the artefacts being restructured are affected by refactoring. When software is restructured, several metric values change and it is interesting to know in advance, the effects of each transformation (KATAOKA et al., 2002). An approach to help in this quantitative evaluation is valuable.

After computing quantitatively the effects of the refactoring pattern application on

software quality, the fourth issue focuses on the qualitative evaluation that the proposed application of refactoring patterns are advantageous in terms of quality. The developer can analyse the effects of each proposed application of a refactoring pattern to choose which ones he will apply.

In the context of refactoring, a shortcoming is a deficiency, inadequacy or incompleteness that a software element can have. Shortcomings typically affect negatively the quality attributes of a software system. In the context of software quality improvement for object-oriented software there are plenty of catalogues of shortcomings for object-oriented software (OPDYKE, 1992; FOWLER et al., 1999). These catalogues can be used to help in the search for refactoring opportunities.

At the beginning of the research work leading to this thesis there were no available catalogues of shortcomings on aspect-oriented software (KICZALES et al., 1997), neither mechanisms to automatically detect their occurrences in aspect-oriented programming languages.

Additionally to the use of catalogues of shortcomings, the developer can use metrics to quantitatively evaluate the software. Metrics adapted from widely known and used metrics for object-oriented software (CHIDAMBER; KEMERER, 1994) have already been used in experimental studies on aspect-oriented software development (CACHO et al., 2006; CASTOR FILHO; GARCIA; RUBIRA, 2005; GREENWOOD; BLAIR, 2006), and the original object-oriented metrics were extended to be independent of the paradigm, generating comparable results (CASTOR FILHO et al., 2006). Up to date, these metrics have been informally described (CECCATO; TONELLA, 2004), but their properties have not been analysed and typical values of these metrics for actual and practical software are not yet available in the literature. Such lack of information about the shape of aspect-oriented software, in terms of each of those metrics, makes their use hard.

1.2 Contributions

The main contributions of this thesis are described as follows:

- **Activities for a Refactoring Process.** A discipline is defined, containing the main activities, roles, and artefacts needed for refactoring. The activities are driven by a quality model selected by the development team. The discipline includes support for (i) the selection of quality models, (ii) the selection and ranking of refactoring patterns, (iii) the selection and creation of heuristic rules, (iv) the search for refactoring opportunities, (v) the computation of the effects of refactoring on software quality, (vi) trade-off analysis and prioritisation of refactoring opportunities and (vii) the application of refactoring patterns. Such refactoring activities can be integrated into a software development process, to provide support to apply refactoring patterns to software elements in several phases of the process.

- **A Method for Ranking Refactoring Patterns.** This thesis proposes a method to rank refactoring patterns according to the quality attributes of a quality model. The relative importance of each quality attribute over the others and the relative importance of the refactoring patterns over quality attributes is quantitatively expressed using the Analytical Hierarchy Process (AHP) multi-criteria decision method (SAATY, 1990, 2003). Using this quantified information, a ranking of refactoring patterns in terms of their contribution to ranked quality attributes can be

automatically computed. The use of such ranking can optimise the search for refactoring opportunities, enabling the developer to focus on refactoring patterns that contribute most to improve the required quality attributes of a piece of software.

- **A Catalogue of Shortcomings in Aspect-Oriented Software.** This catalogue focus on shortcomings that can occur in aspect-oriented software, in such a way to (i) describe the problems that arises whenever those shortcomings are present in aspects and (ii) propose the use of refactoring patterns to help to minimize or remove them. It also focuses on the automatic detection of shortcomings in programs written in the *AspectJ* language (KICZALES et al., 2001a). The main goal is to provide a prototype implementation to detect a sub-set of the shortcomings described in the catalogue.

- **An Approach for the Search of Refactoring Opportunities.** An approach is proposed for the search of refactoring opportunities, allowing developers to focus on the refactoring opportunities that maximize the quality attributes they are interested in. Heuristic rules are used to evaluate the software elements according to a selected quality model. Qualitative analysis is conducted and the trade-offs for each opportunity are analysed. The developer can order and filter the available refactoring opportunities, and can discard unfruitful opportunities or mark specific ones for refactoring.

- **An Approach to Evaluate the Effects of Refactoring on Software Quality.** A set of activities are proposed to evaluate quantitatively how each application of a refactoring pattern affects software quality. The quality of software is evaluated using heuristic rules and impact functions. Heuristic rules are mathematical functions created according to a quality model and its associated metrics, and are used to spot shortcomings in software artefacts. Impact functions predict the values of metrics simulating the application of a refactoring pattern. With heuristic rules and impact functions, there is no need to apply each refactoring pattern to evaluate its effects.

- **An Approach to Reduce the Number of Refactoring Sequences.** This thesis proposes an approach to reduce the number of refactoring sequences to be analysed for a software application. This reduction of sequences includes the simplification of equivalent, commutative, inverse, impossible and parallel sequences. This simplification can help the developer to focus on searching for refactoring opportunities for the most promising refactoring sequences. The proposed approach is representation independent and can be used together with different formalisms to handle refactoring sequences.

- **A Suite of Metrics for Aspect-Oriented Software.** This work complements previous works on metrics for aspect-oriented software and for object-oriented software through the formal definition of these metrics, their analytical evaluation, and detailed discussion about empirical data collected from a set of applications, providing guidelines about the typical values of the metrics. The formal definitions of the metrics allow expressing unambiguously how each metric is computed. Empirical data show how typical aspect-oriented software are organised and how the metric values are distributed in different projects. Usage guidelines show how the metrics can be used and analytical evaluation complements the formal definitions of the metrics by evaluating a set of properties expected from the metrics.

1.3 Evaluation

The evaluation of each contribution is described as follows:

- **Activities for Refactoring.** Several activities, roles and artefacts needed to include refactoring in software development processes are described. These activities are exemplified for object-oriented software and aspect-oriented software throughout the thesis, where common sequences of activities are exercised. Note that there is still the need to apply the process in large-scale projects, in different languages, different teams and different domains to be fully aware of its implications. The individual activities are evaluated individually, as previously described.

- **A Method for Ranking Refactoring Patterns.** The method is evaluated through the creation of a ranking of refactoring patterns, exemplified using three quality attributes and four refactoring patterns. Although the example is based on refactoring patterns for object-oriented software, the approach can be generalised to be used in other paradigms, and other artefacts besides source code. An API for computing such rankings was developed to help the developer on automating this task.

- **A Catalogue of Shortcomings in Aspect-Oriented Software.** The catalogue describes each type of shortcoming, including refactoring patterns that can be used to minimize its effect, examples in AspectJ programs and rules which can be used to detect them. Tool support is provided and is used in a case study comprised of three open source aspect-oriented programs.

- **An Approach for the Search for Refactoring Opportunities.** The approach is exemplified for an object-oriented software application. Two quality attributes are assessed using a heuristic rule, and the search is exemplified using two refactoring patterns in a software application composed of 200 classes.

- **An Approach to Evaluate the Effects of Refactoring on Software Quality.** The approach is exemplified by defining impact functions for an aspect-oriented refactoring pattern, named *Pull Up Advice*, for four metrics: lines of code, number of operations in module, crosscutting degree of an aspect and coupling on advice execution. Tool support is provided to automate the computation of metrics and the expected changes in the heuristic rules values.

- **An Approach to Reduce the Number of Refactoring Sequences.** The approach is exemplified through the use of deterministic finite automata (DFA) (SIPSER, 1996) to represent refactoring sequences for method manipulation and a set of simplification rules to reduce the number of sequences. It is shown that the number of sequences can be greatly reduced by simplification (the example showed a 62% reduction of the number of sequences). Tool support is included to generate the initial sequences.

- **A Suite of Metrics for Aspect-Oriented Software.** The evaluation is conducted using (a) formal definitions and usage scenarios for the selected metrics, (b) an interpretation of collected empirical data, discussing minimum and maximum values, comparing the values in aspects and in classes and examining variations between the metric values of the selected projects and (c) an analytical evaluation of the selected metrics against established criteria.

1.4 Thesis Outline

This thesis is organised as follows.

Chapter 2 details the main concepts needed to understand the thesis approach and evaluation. It discusses the following themes: refactoring, aspect-oriented software development (AOSD), and the Analytical Hierarchy Process (AHP).

Chapter 3 describes the activities required to include refactoring activities in a software development process. It starts by highlighting the main activities to be performed and the associated roles and artefacts. Each activity is detailed to show its value in the overall process structure.

Chapter 4 describes how a set of refactoring patterns can be ranked to improve a set of quality attributes of a piece of software. AHP is used to express the relative importance of the quality attributes and the relative importance of refactoring patterns in regards to those selected quality attributes.

Chapter 5 describes a collection of shortcomings which occur in aspect-oriented software, while Chapter 6 shows how metrics can be grouped together as heuristic rules and how they can be used to prioritise opportunities for refactoring in software elements, aiming at improving their quality by focusing on the opportunities that are more likely to produce positive effects in a selected quality model. Chapter 6 also includes rules and proof-of-concept implementations for a sub-set of shortcomings for aspect-oriented software.

Chapter 7 presents a rationale for the creation of impact functions for refactoring object-oriented software and aspect-oriented software. This rationale can be used to create impact functions for software metrics or heuristic rules and is demonstrated through a detailed example, and a case study.

Chapter 8 describes an approach to reduce the number of sequences of refactoring patterns. First, it is described how a set of initial sequences can be created. Secondly, these sequences are simplified using a set of simplification rules.

Chapter 9 shows formal definitions and usage guidelines of of six metrics for aspect-oriented software. It also includes empirical data collected from ten open source projects, and a set of examples.

Chapter 10 discusses the main conclusions of this thesis, including a summary of the contributions and proposed research areas for future work.

Appendix A describes the main papers accepted during the thesis development. Appendix B proposes a set of guidelines to reduce the occurrence of shortcomings in aspect-oriented software artefacts and exemplifies the benefits of using the described guidelines.

Appendix C shows an analytical evaluation of six selected metrics against established criteria. The chapter shows that the aspect-oriented metrics also satisfy the criteria originally satisfied by their counterpart object-oriented metrics, recommending that they can be used to assess aspect-oriented software.

Appendix D shows the computations of a set of impact functions for the *Pull Up Advice* refactoring pattern in an example application. Appendix E shows impact functions for six refactoring patterns for object-oriented software and how they can be composed to specify other impact functions. Appendix F shows impact functions for the *Extract Pointcut* and the *Inline Inter-Type Declaration* refactoring patterns.

2 BACKGROUND

This chapter describes a set of background areas, which help to understand the main issues of this thesis. It is organised as follows. Section 2.1 describes the state-of art of refactoring. Section 2.2 describes the main concepts of aspect-oriented software development (AOSD) and approaches for refactoring aspect-oriented software. Section 2.3 describes the Analytical Hierarchical Process (AHP), a multi-criteria decision method, which is used in this thesis as a method for the selection and ranking of refactoring patterns.

2.1 Refactoring

Refactoring (OPDYKE, 1992; FOWLER et al., 1999; MENS; TOURWE, 2004) is the process of improving the design of software systems without changing their externally observable behaviour. Refactoring can help to incrementally improve the quality attributes (ISO, 2001; BOEHM; IN, 1996) of a software system through the application of behavioural preserving transformations called *refactoring patterns* (KATAOKA et al., 2001).

Refactoring patterns are organised into collections of patterns called *refactoring patterns catalogues*. Each refactoring pattern in a catalogue is described by a name, a context in which it should be applied, a set of well-defined steps for its application, and one or more examples showing how the transformation should occur (FOWLER et al., 1999).

Examples of typical refactoring patterns are those for manipulating classes, methods and fields, such as: extraction of a sub-class, extraction of a super-class, movement of methods, encapsulating fields, and inlining methods. Table 2.1 shows a set of commonly used refactoring patterns in the context of object-oriented software.

The efforts on research in the context of refactoring are grouped into four groups: (a) identifying where to apply refactoring patterns, (b) assessing the effects of refactoring on quality, (c) guaranteeing that the refactoring patterns preserve software behaviour, and (d) maintaining consistency of the refactored software artefacts (MENS; TOURWE, 2004). The last two activities (consistency maintenance and guaranteeing behavioural preservation) are outside of the scope of this thesis.

Section 2.1.1 discusses the identification of where to apply refactoring patterns. Section 2.1.2 describes research focusing the assessment of the effects of refactoring on quality.

Table 2.1: Examples of refactoring patterns for object-oriented software

Refactoring Pattern	**Description**	**Source**
Chain Constructors	Chains a set of constructors together to obtain a low amount of duplicated code.	(KERIEVSKY, 2005)
Collapse Hierarchy	Merges a super class and a subclass together.	(FOWLER et al., 1999)
Delete Attribute	Deletes an attribute that is not being referenced by any class.	(OPDYKE, 1992)
Delete Class	Deletes a class and all references to it.	(OPDYKE, 1992)
Delete Method	Deletes a method that is not being referenced by any class.	(OPDYKE, 1992)
Encapsulate Attribute	Creates accessors for the attribute and replaces all the read and write operations to the attribute by calls to the accessors.	(FOWLER et al., 1999)
Extract Class	Creates a new class and moves to it fields and methods from an existing class.	(FOWLER et al., 1999)
Extract Interface	Extracts method signatures and creates an interface for a class.	(FOWLER et al., 1999)
Extract Method	Extracts a piece of code to a new method.	(FOWLER et al., 1999)
Inline Class	Moves all the features of a chosen class into another class. Deletes the chosen class.	(FOWLER et al., 1999)
Inline Method	Replaces all the method calls of a specific method by the contents of that method.	(FOWLER et al., 1999)
Inline Sigleton	Moves the features of a Singleton to a class that stores and provides access to the object. Deletes the Singleton.	(KERIEVSKY, 2005)
Introduce Explaining Variable	Extracts an expression to a local variable.	(FOWLER et al., 1999)
Move Attribute	Moves an attribute to another class.	(FOWLER et al., 1999)
Move Embellishment to Decorator	Moves an embellishment code to a Decorator.	(KERIEVSKY, 2005)
Move Method	Moves a method from one class to another.	(FOWLER et al., 1999)
Pull Up Attribute	Moves an attribute to a super-class or super-aspect of the current class or aspect.	(FOWLER et al., 1999)
Pull Up Method	Moves a method to a super-class of the current class.	(FOWLER et al., 1999)
Push Down Method	Moves a method to one or more of its subclasses.	(FOWLER et al., 1999)
Rename Class	Changes the name of a class and in all the places that it is referenced.	(FOWLER et al., 1999)
Rename Method	Changes the name of a method and all the method calls to a new name.	(FOWLER et al., 1999)
Replace Method with Method Object	Extracts a method from a set of selected statements to a new class, containing the extracted statements as a new method and the local variables of the method as fields of the new class.	(FOWLER et al., 1999)

2.1.1 Identification of Refactoring Opportunities

In the context of refactoring, a *shortcoming* is a deficiency, inadequacy or incompleteness that a software element can have. Shortcomings typically affect negatively the quality

attributes of a software system. There are catalogues of shortcomings in the context of both object-oriented and aspect-oriented software development.

For example, Fowler et al. (FOWLER et al., 1999) describe 22 shortcomings in the context of object-oriented software. Kerievsky (KERIEVSKY, 2005) catalogs 12 shortcomings which motivate the application of *refactoring to patterns* (refactoring patterns that, when applied, changes a given design to use a specific design pattern). Monteiro and Fernandes (MONTEIRO; FERNANDES, 2006) describe three shortcomings for aspect-oriented code, and Srivisut and Muenchaisri (SRIVISUT; MUENCHAISRI, 2007) describe a set of additional shortcomings that can be found in aspect-oriented software, algorithms to automate their detection and suggested refactoring patterns to remove them.

There are some approaches for identifying occurrences of shortcomings in software applications. Simon et al. propose a metric based approach (SIMON; STEINBRUCKNER; LEWERENTZ, 2001) to identify opportunities for the application of four refactoring patterns: *Move Method*, *Move Attribute*, *Extract Class* and *Inline Class*. They plot the methods and attributes of the classes in a graphical representation and the developer decides which methods and attributes will be moved based on the cohesion of attributes and methods. An equation is presented to evaluate the cohesion of methods and attributes within the classes of a software application. The results are converted to a three-dimensional Cartesian coordinate system, and then rendered visually. Similar approaches are used by Lanza and Ducasse (LANZA; DUCASSE, 2002) and by van Emden (EMDEN, 2002).

Tourwe and Mens use logic meta-programming (TOURWE; MENS, 2003) (which is meta-programming using the logic paradigm) to search for refactoring opportunities in existing software. They define a framework that uses meta-information of software programs to propose the application of refactoring patterns. Logic programming statements are used to detect occurrences of shortcomings in software applications.

The tactics of Balazinska et al. (BALAZINSKA et al., 2000) and Ducasse et al. (DUCASSE; RIEGER; DEMEYER, 1999) are similar in that both attempt to find repeated sections of source code throughout a software application. Balazinska's approach focuses on Java code and thus involves the parsing of the code, while Ducasse's approach tries to remain language independent, considering the source code only as text strings. A few other approaches to automate the detection of shortcomings in software are presented by Mens and Tourwe (MENS; TOURWE, 2004).

2.1.2 Assessing the Effects of Refactoring on Quality

An open problem is to assess the effects of a refactoring pattern on the quality of a software system, as some refactoring patterns remove redundancy, raise abstraction or modularity level and others have negative impact on reusability, for example (MENS et al., 2003). By classifying refactoring patterns in terms of the quality attributes they affect, the effects of a refactoring on the software quality can be estimated.

A formalism proposed to describe the impact of refactoring patterns uses the AST representation of the source code (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005), extended with cross-references. This formalism uses AST representing object-oriented programs and metrics and is used to evaluate the impact of *Extract Method*, *Encapsulate Attribute* and *Pull Up Method* refactoring patterns and describe how quality metrics can be defined on top of this program structure representation.

In this approach (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006) coupling and cohesion metrics are used to identify the conditions under which the

application of some refactoring patterns minimize coupling and maximize cohesion. A set of functions to evaluate the impact of the *Extract Method*, *Move Method* and *Replace Method with Method Object* refactoring patterns on software quality are formally defined. Another approach (MOSER et al., 2006) uses a set of metrics to evaluate the quality of source code to analyse the impact of refactoring on reusability. A mathematical function is defined to measure reusability quantitatively.

Other approaches deal with successive refactoring (i.e. the successive application of refactoring patterns) to improve software quality. Liu et al. (LIU et al., 2007) propose an approach to deal with successive refactoring by providing a model for dealing with conflicting refactoring patterns and a heuristic rule to solve that model. The problem of structural evolution conflicts is handled in a formal way by using graph transformations and critical pair analysis (MENS; TAENTZER; RUNGE, 2005), which are used to detect and resolve refactoring conflicts.

2.2 Aspect-Oriented Software Development

Some software requirements are mapped onto concerns that affect several classes in a systematic manner, modifying the semantics of the classes and/or the performance of an application. Examples of these concerns: persistence support, debugging, and distribution. With object-oriented languages, their implementation is usually done in a way that causes code fragments corresponding to them to be dispersed along several application classes. Fields and methods that handle these properties appear in several classes, either directly (inside the class) or indirectly (through associations) (KICZALES et al., 1997).

The implementation of such requirements using object-oriented techniques can cause problems in comprehending the functionality of the classes, as well as in the reusability and maintainability of a software system (LOPES, 1997). Aspect-oriented software development allows the developer to use a new abstraction mechanism, called *aspect*, to separate the functional components of the application from these requirements.

Aspects alter classes and other aspects through static and dynamic mechanisms. The static mechanisms add data and behaviour to classes, while the dynamic mechanisms modify the software system at runtime.

Consider the following example: the implementation of a logging agent that stores information to be used in case of execution problems or unauthorized access attempts. Figure 2.1 shows such implementation in the Apache TomCat JSP container. In this figure, the classes are represented by white and grey lines. Classes that are not affected by logging are greyed and the highlighted bars represent every call to code relating to logging (HILSDALE; KICZALES, 2001).

From this global view, it is possible to see how difficult is to modify logging policies in the application. Replacing the logging framework or its configuration (to remove development-time test entries, for example) can be an exhausting task, since the logging code is scattered throughout application classes and tangled with the code.

The aspect-oriented approach aims at separating cross-cutting concerns in first class entities called aspects. These aspects define which points in the software application will be affected and what happens with the application whenever these points are reached. Figure 2.2 shows how the same concern appears when encapsulated in a logging aspect.

Other system abstractions are not necessarily aware of the existence of such aspects. This property of aspect-oriented software systems is called obliviousness (FILMAN; FRIEDMAN, 2000), and is interesting (but not essential) for the effective use of aspects.

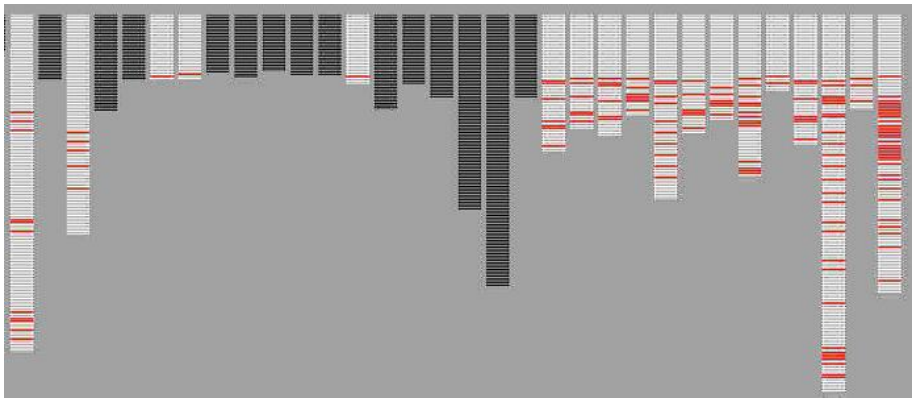

Figure 2.1: *Logging* in *org.apache.tomcat*

Figure 2.2: *Logging* in *org.apache.tomcat* using aspects

The main goal of aspect-oriented software development is to help in the task of separating cross-cutting concerns. This is done by the use of abstraction and composition mechanisms that support clear separation and composition of cross-cutting concerns. Aspect-oriented software development extends other techniques (object-oriented, structured, and functional) that provide no abstractions specifically designed to deal with cross-cutting concerns (KICZALES et al., 1997).

2.2.1 AspectJ

AspectJ is an aspect-oriented language based on the Java programming language. Besides the usual object-oriented abstractions (classes, methods, and fields), the language provides abstractions related to aspects implementation, such as: pointcuts, join points, advices and inter-type declarations. These concepts are explained in the following sections.

2.2.1.1 Join Points

Join points are well-defined points in the execution flow of a program. Examples of join points are: method and constructor calls and execution, field access and initializations. Consider, for example, an *Account* class, containing a method named *withdraw* (representing the withdrawal of money) and a field named *balance* (to store the balance of the account).

```
1 public class Account {
```

```
2     double balance;
3     public void withdraw(double value){
4       balance -= value;
5     }
6     public static void main(){
7       Account c = new Account();
8       c.balance = 100;
9       c.withdraw(50);
10    }
11  }
```

In this context, join points would be, for instance, the execution of the *withdraw* method (line 4), its call on line 9, reading/writing the *balance* field (lines 4 and 8), and the *Account* object instantiation (line 7). In AspectJ, there are syntactic elements that allow to describe join points representing the affected points in the software application.

2.2.1.2 Pointcuts

Pointcuts group join points by the definition of a predicate that, whenever satisfied, causes the advices associated to it to be executed. Several elements can be used to define them. These join points can also be composed using the logical operators *and*, *or* and *not* (&&, || and ! respectively).

Pointcuts can be named and receive parameters. These represent the formal arguments of the pointcut, for example, the object that receives the message, the current object and the actual parameters. They can be inspected and modified according to the expected behaviour of the aspect.

For example, to define an aspect that performs some actions whenever a call to the *Account.withdraw* method is made, a pointcut can be defined as follows: *call(void Account.withdraw(double))*. To define that the aspect affects the creation of new objects a developer can use: *initialization(public Account.new(..))*. The AspectJ pointcut language is very powerful, and enables to describe join points both in the static structure as in the dynamic structure of a software system.

2.2.1.3 Inter-type declarations

Inter-type declarations introduce state or behaviour to an existing class, aspect or interface. Three kinds of declarations are commonly used: inter-type fields, inter-type methods, and inter-type constructor declarations. They respectively introduce new fields, methods, and constructors to an existing module.

2.2.1.4 Advice

An advice is an action associated to a pointcut. These actions can occur *before*, *after* or *around* a join point (this is determined by different keywords used). The *after* advice can still have two variations: it may be executed after the successful run of the code associated to the pointcut, or in cases where an exception occurs while executing the advice code.

2.2.1.5 Aspects

In AspectJ, there is a new abstraction named *aspect*, declared using the keyword `aspect`. Aspects are similar to classes in several ways: they can contain fields, methods, and implement interfaces. However, unlike classes, they cannot be instantiated, and

their inheritance mechanism is somewhat limited (only abstract aspects or classes can be extended). Aspects unite pointcuts, join points, inter-type declarations and advices in a single abstraction mechanism.

2.2.2 Measuring Aspect-Oriented Software

When measuring aspects, the developer can focus on the relation of the aspect with other modules (aspects, classes or interfaces) in terms of the use of inheritance (number of children, depth of inheritance tree), associations (coupling metrics), and affected modules (crosscutting degree of an aspect). Also, the developer can measure aspects members (advices, pointcuts, inter-type declarations). For example, advices can be measured in terms of their code size (lines of code) or the number of modules that a particular advice affects. Pointcuts can be measured to evaluate the size and complexity of pointcut expressions and also to determine the number of join points the expression affects. Inter-type declarations can be used as a part of the measurements to express the complexity of an aspect (such as the number of operations in module metric).

Previous works on metrics applicable to aspect-oriented software are typically extensions of object-oriented metrics (CHIDAMBER; KEMERER, 1994). In fact, some of these metrics were revisited to take in account the specific features of aspect-oriented software. Castor et al. (CASTOR FILHO; GARCIA; RUBIRA, 2005) propose a suite of metrics, including metrics for separation of concerns, coupling, cohesion and size. This suite has already been used in some experimental studies (CACHO et al., 2006; CASTOR FILHO; GARCIA; RUBIRA, 2005; GREENWOOD; BLAIR, 2006).

The metrics included in this suite can be briefly summarized as follows:

- *Lines of Class Code (locc):* Counts the lines of code.
- *Number of Attributes (noa):* Counts the number of fields of each class or aspect.
- *Number of Operations (noo):* Counts the number of methods and advices of each class or aspect.
- *Concern Diffusion over Components (cdc):* Counts the number of components that contribute to the implementation of a concern and other components which access them.
- *Concern Diffusion over Operations (cdo):* Counts the number of methods and advices that contribute to the implementation of a concern plus the number of other methods and advice accessing them.
- *Concern Diffusion over locc (cdl):* Counts the number of transition points (points in the code where there is a *concern switch*) for each concern through the lines of code.
- *Coupling Between Components (cbc):* Counts the number of components declaring methods or fields that may be called or accessed by other components.
- *Depth of Inheritance Tree (dit):* Counts how far down in the inheritance hierarchy a class or aspect is declared.
- *Lack of Cohesion in Operations (lco):* Measures the lack of cohesion of a class or aspect in terms of the amount of method and advice pairs that do not access the same field.

Other authors (CECCATO; TONELLA, 2004) also discuss metrics to count the number of operations, the depth of inheritance tree (dit) and lack of cohesion in operations (lco). Other metrics include the following:

- *Number of Children (noc):* Number of immediate sub-classes or sub-aspects of a given module, indicating the proportion of modules potentially dependent on inherited properties.

- *Coupling on Advice Execution (cae):* Number of aspects containing advices possibly triggered by the execution of operations in a given module.

- *Crosscutting Degree of an Aspect (cda):* Number of modules affected by the pointcuts and by the inter-type declarations of a given aspect.

- *Coupling on Intercepted Modules (cim):* Number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect.

- *Coupling on Method Call (cmc):* Number of modules or interfaces declaring methods that are possibly called by another given module.

- *Coupling on Field Access (cfa):* Number of modules or interfaces declaring fields that are accessed by another given module.

- *Response for a Module (rfm):* Methods and advices potentially executed in response to a message received by a given module, measuring the potential communication between this module and the other ones.

Chapter 9 describes a set of metrics used in this thesis to express shortcomings in aspect-oriented software.

2.2.3 Refactoring Aspect-Oriented Software

Existing refactoring catalogues (OPDYKE, 1992; FOWLER et al., 1999) define and describe several refactoring patterns that can be used to improve the design of object-oriented software. Among these are refactoring patterns to reorganise hierarchies, to move methods for other classes, and to encapsulate attributes.

In the context of aspect-oriented software, there is the need of refactoring patterns that allow the manipulation of both classes and aspects. Specifically, refactoring patterns that deal with aspect-oriented software must allow moving code: (a) from classes to aspects, (b) between aspects and (c) from aspects to classes.

Several refactoring patterns have been proposed to enable the manipulation of software elements in aspect-oriented software (GARCIA et al., 2004; HANENBERG; OBERSCHULTE; UNLAND, 2003; IWAMOTO; ZHAO, 2003; MONTEIRO; FERNANDES, 2004, 2005a). These refactoring patterns help to remove or minimize the occurrence of shortcomings in aspect-oriented code.

Table 2.2 lists a set of refactoring patterns for aspect-oriented software. For each refactoring pattern are provided its name, a description and the original reference.

Iwamoto and Zhao (IWAMOTO; ZHAO, 2003) present an analysis of several refactoring patterns from Fowler's refactoring catalogue (FOWLER et al., 1999), concluding that few of them can be used in aspect-oriented code without modifications. Iwamoto and

Table 2.2: Examples of refactoring patterns

Refactoring Pattern	**Description**	**Source**
Add Aspect Precedence	Adds a declare precedence construction to an aspect.	(MONTEIRO; FERNANDES, 2006)
Collapse Aspect Hierarchy	Merges an aspect hierarchy	(GARCIA et al., 2004)
Combine Pointcut	Unites the predicates of several pointcuts	(IWAMOTO; ZHAO, 2003)
Convert Aspect to Class	Converts an aspect to a class. Advices, pointcuts and inter-type declarations must be inlined or moved to another aspect.	(MONTEIRO; FERNANDES, 2006)
Delete Aspect	This refactoring pattern deletes an aspect and all references to it.	(MONTEIRO; FERNANDES, 2006)
Extract Aspect	Creates a new aspect with selected members from an existing class or aspect	(MONTEIRO; FERNANDES, 2006)
Extract Pointcut	Extracts a pointcut definition from an advice	(IWAMOTO; ZHAO, 2003)
Extract Sub-Aspect	Creates an sub-aspect containing a subset of features	-
Inline Aspect	Inserts the code from aspects to the classes it affects	(GARCIA et al., 2004)
Move Advice	Moves an advice from from a source aspect to a destination aspect.	(GARCIA et al., 2004)
Move Inter-Type Declaration	Moves an inter-type declaration from a source aspect to a destination aspect.	(MONTEIRO; FERNANDES, 2006)
Move Pointcut	Moves a pointcut from one class/aspect to another	-
Pull Up Advice	Moves an advice to a super-class or super-aspect of the current aspect	(GARCIA et al., 2004)
Pull Up Inter-type Declaration	Moves an inter-type declaration to a super-class or super-aspect of the current aspect	(GARCIA et al., 2004)
Pull Up Pointcut	Moves a pointcut to a super-class or super-aspect of the current aspect	(GARCIA et al., 2004)
Remove Advice Parameter	Removes a parameter from an advice	-
Rename Aspect	Renames an aspect and every reference to it	(HANENBERG; OBERSCHULTE; UNLAND, 2003)
Rename Pointcut	Renames a pointcut and every reference to it	(GARCIA et al., 2004)

Zhao also define several refactoring patterns in the context of aspects, including refactoring patterns for extraction of advices and pointcuts, and the creation of pointcuts and advices.

Hanenberg et al. (HANENBERG; OBERSCHULTE; UNLAND, 2003) discuss the re-

lationship between refactoring patterns for aspect-oriented and object-oriented software, describing the conflicts found and suggesting ways to solve them in *AspectJ*. They also introduce new refactoring patterns to help migrating object-oriented code to aspects, as well as restructuring aspect code.

Garcia et al. (GARCIA et al., 2004) describe a set of interrelated refactoring patterns to handle cross-cutting concerns. Some refactoring patterns defined by the authors aim at manipulating aspect-oriented code, such as: *Rename Pointcut*, *Collapse Aspect Hierarchy* and *Collapse Pointcut Definition*.

Monteiro and Fernandes (MONTEIRO; FERNANDES, 2004, 2005a, 2006) present a catalogue of refactoring patterns to help in aspect extraction from legacy object-oriented code. The authors discuss shortcomings that might appear in aspect-oriented software systems, including one which occurs only in aspects. The same authors (MONTEIRO; FERNANDES, 2005b) also emphasize the importance of the development of both refactoring patterns and catalogues of shortcomings.

Zhang et al. (ZHANG et al., 2005) define a tool to verify aspects restructured by the application of refactoring patterns against the original source code. They use aspect mining and aspect exploration information as the reference for verification. The tool is implemented as an *Eclipse* plug-in working together with the AJDT tool (*AspectJ Development Tool*).

Deursen et al. (DEURSEN; MARIN; MOONEN, 2005) propose a common show case for the application of aspect-oriented techniques. They suggest that the *HotDraw* framework (JOHNSON, 2002) can be used as a show case for refactoring techniques and for the detection and removal of shortcomings.

2.3 The Analytical Hierarchy Process

The Analytical Hierarchy Process (AHP) (SAATY, 1990) is a mathematical decision making technique for evaluating a set of different alternative solutions of a given problem. It focuses on finding an optimal solution using qualitative and quantitative decision analysis. In this thesis, AHP is used to express the multiple criteria used to decide if the application of a refactoring pattern is interesting in terms of the satisfiability of the quality attributes specified for the software system, and to rank refactoring patterns according to a quality model.

AHP comprises of five steps: (a) definition of the problem and its objective, (b) hierarchical representation, (c) estimation of priorities, (d) synthesis and (e) results consistency analysis.

Section 2.3.1 describes the first two steps (problem definition and hierarchical representation). Section 2.3.2 shows how the estimation of priorities is conducted. Section 2.3.3 exemplifies the synthesis step and Section 2.3.4 details how the consistency index is computed and how its value should be analysed.

2.3.1 Problem Definition and Hierarchical Representation

The *definition of the problem and its objective* establishes the context in which the decision making process will occur. After that, the problem is *hierarchically represented*, with the objective having several associated criteria ($C_1, \ldots, C_n$) and each criterion several alternatives ($A_1, \ldots, A_n$), as shown in Figure 2.3.

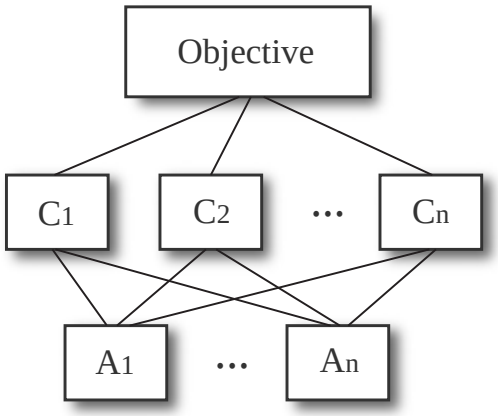

Figure 2.3: AHP hierarchical problem representation

2.3.2 Priorities Estimation

In the *priorities estimation* step, the developer defines the priorities for the criteria and alternatives. The relative importance of each criterion over the others and each alternative over the others is ascertained using pairwise comparisons. The scale in Table 2.3 (SAATY, 1990) is used to numerically express the relative importance over criteria and alternatives.

Table 2.3: Numerical values for the relative importances.

Value	Relative Importance
1	Same importance
2	Slightly more important
3	Weakly more important
4	Weakly to moderately more important
5	Moderately more important
6	Moderately to strongly more important
7	Strongly more important
8	Greatly more important
9	Absolutely more important

Using this process, a pairwise matrix can be created, containing the relative importance of each criterion over the others. Consider, for example, a set of criteria $\mathcal{C} = \{c_1, \ldots, c_n\}$ and a matrix $\mathcal{M}$ representing the relative importance of each criterion over the others $W = \{w_{11}, w_{12}, \ldots, w_{nn}\}$. The pairwise matrix in this case, is constructed as follows:

$$\mathcal{M} = \begin{bmatrix} 1 & w_{12} & \cdots & w_{1n} \\ 1/w_{21} & 1 & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 1/w_{n1} & 1/w_{n2} & \cdots & 1 \end{bmatrix} \tag{2.1}$$

2.3.3 Synthesis

The next step of AHP, named *synthesis*, aims at computing a vector containing the relative weights of all the criteria in the matrix. This vector represents the ranking of priorities given by this pairwise matrix. To compute this vector, the matrix is squared successively. In each iteration, the row sums are calculated and normalized. The computation stops when the differences of these sums in two consecutive calculations are smaller than a prescribed value. Usually two to four iterations are sufficient.

To exemplify these last steps, consider three criteria c_1, c_2, c_3 and a pairwise matrix $\mathcal{M}$ containing the importance of each criterion over another:

$$\mathcal{M} = \overbrace{\begin{bmatrix} 1 & 1/2 & 3 \\ 2 & 1 & 4 \\ 1/3 & 1/4 & 1 \end{bmatrix}}^{c_1 \quad c_2 \quad c_3} \begin{matrix} c_1 \\ c_2 \\ c_3 \end{matrix} \tag{2.2}$$

In this case, c_1 is *weakly more important* than c_3, c_2 is *slightly more important* than c_1, and c_2 is *weakly to moderately more important* than c_3.

The ranking of priorities given this pairwise matrix can be derived from the weights vector of the matrix. To compute this vector, the matrix is squared, the sum of the row values is computed and the values are normalized:

$$\overbrace{\begin{bmatrix} 3.0000 & 1.7500 & 8.0000 \\ 5.3333 & 3.0000 & 14.0000 \\ 1.1667 & 0.6667 & 3.0000 \end{bmatrix}}^{squared\ matrix} = \overbrace{\begin{bmatrix} 12.7500 \\ 22.3332 \\ 4.8333 \end{bmatrix}}^{sum\ of\ rows} = \overbrace{\begin{bmatrix} 0.3194 \\ 0.5595 \\ 0.1211 \end{bmatrix}}^{normalized}$$

$$\textbf{Total} \quad 39.9165 \quad 1.0000$$

This normalized vector $V = \langle 0.3194 \ 0.5595 \ 0.1211 \rangle$ is called the eigenvector (SAATY, 2003) of the matrix. The procedure is repeated again, which lead us to the following eigenvector: $V' = \langle 0.3196 \ 0.5584 \ 0.1220 \rangle$. Additional iterations do not change these values (at least using a four digits precision).

Therefore, the weights for the 3-tuple C_s = (c_1, c_2, c_3) are respectively the values of

$$V' = \langle 0.3196 \ 0.5584 \ 0.1220 \rangle$$

In this case, the function to determine how *good* is an alternative *a* in terms of the selected criteria can be expressed as:

$$f(a) = 0.3196 * f_{c1}(a) + 0.5584 * f_{c2}(a) + 0.1220 * f_{c3}(a) \tag{2.3}$$

where f_{c1}, f_{c2}, f_{c3} are functions that quantitatively assess the criteria c_1, c_2 and c_3.

2.3.4 Results Consistency Analysis

The last step of AHP, *results consistency analysis* (SAATY, 1990), evaluates the consistency level of the pairwise comparison matrix. This is necessary because the pairwise comparison may insert inconsistencies in the process, since the relative importance of one criterion over another is the result of several judgements about the relationship of the criteria and those judgements can conflict with one another.

Consider for example, that the person who specifies the matrix says that c_1 is more important than c_2, c_2 is more important than c_3, and c_3 is more important than c_1. In this case at least one of the judgements done through pairwise comparisons is not accurate.

The results consistency analysis verifies whether the existing inconsistencies in the comparisons are acceptable or not. The process is considered acceptable if the consistency ratio (*CR*) is below 10%.

The consistency ratio (*CR*) can be computed as follows:

$$CR = \frac{CI}{RCI} \tag{2.4}$$

where *CI* is the first precision degree of the pairwise comparisons and *RCI* is the consistency degree obtained in pairwise comparisons in a set of random pairwise matrices. The details of the *RCI* calculation are discussed by Saaty (SAATY, 1990), who provides a *RCI* table to be used in the consistency ratio calculation (see Table 2.4).

Table 2.4: Random consistency index - reference values

n	< 3	3	4	5	6	7	8
RCI	0	0.58	0.9	1.12	1.24	1.32	1.41
n	9	10	11	12	13	14	15
RCI	1.45	1.49	1.51	1.48	1.56	1.57	1.59

The *CI* value is given by:

$$CI = \frac{\lambda_{max} - n}{n - 1} \tag{2.5}$$

where λ_{max} is the maximum eigenvalue of the comparison matrix.

The matrix described in Equation 2.2 and its eigenvector (2.6) are used to exemplify the consistency ratio calculation.

$$V' = \langle 0.3196 \;\; 0.5584 \;\; 0.1220 \rangle \tag{2.6}$$

To obtain this λ_{max}, the following steps are executed:

1. The eigenvector values are multiplied by the columns of the matrix

$$0.3196 * \begin{bmatrix} 1 \\ 2 \\ 1/3 \end{bmatrix} + 0.5584 * \begin{bmatrix} 1/2 \\ 1 \\ 1/4 \end{bmatrix} + 0.122 * \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} \tag{2.7}$$

2. The values of the rows of the resulting vectors are added

$$\begin{bmatrix} 0.3196 \\ 0.6392 \\ 0.1065 \end{bmatrix} + \begin{bmatrix} 0.2792 \\ 0.5584 \\ 0.1396 \end{bmatrix} + \begin{bmatrix} 0.366 \\ 0.488 \\ 0.122 \end{bmatrix} = \begin{bmatrix} 0.9648 \\ 1.6856 \\ 0.3681 \end{bmatrix} \tag{2.8}$$

3. The λ_{max} is the mean of the division of the sum values vector by the values of the eigenvector

$$\lambda_{max} = \frac{\frac{0.9648}{0.3196} + \frac{1.6856}{0.5584} + \frac{0.3681}{0.122}}{3} = 3.0183 \tag{2.9}$$

The *CI* can be calculated as:

$$CI = \frac{\lambda_{max} - n}{n - 1} = \frac{3.0183 - 3}{3 - 1} = 0.0091 \tag{2.10}$$

According to Table 2.4, the *RCI* $= 0.58$ when $n = 3$. Therefore, the consistency ratio *CR* is:

$$CR = 0.0091/0.58 = 0.0158 \tag{2.11}$$

The consistency ratio in the example is inferior to the 10% threshold value representing the acceptable values for consistency when using pairwise comparisons in AHP.

Chapter 6 describes the use of AHP to define the weights for a heuristic rule used to help in the identification of refactoring opportunities in software applications. Chapter 4 describes the use of AHP to rank refactoring patterns to improve a set of quality attributes of a piece of software.

3 A DISCIPLINE FOR REFACTORING

This chapter describes a set of activities to help in the refactoring process. It is organised as follows. Section 3.1 presents an overview of the main activities in order to show how the activities are related and which is their recommended order. Section 3.2 describes activities, roles and artefacts related to the selection and creation of quality models, which drive the refactoring process. Section 3.3 focuses on how a developer can select and rank a set of refactoring patterns according to a quality model. Section 3.4 shows the creation of heuristic rules to search for refactoring opportunities. Section 3.5 describes some issues regarding the search for refactoring opportunities matching a set of pre-defined refactoring patterns, heuristic rules and additional search criteria. Section 3.6 discusses how the effects of refactoring on software quality can be quantitatively computed while Section 3.7 shows how to analyse the proposed opportunities for refactoring in a qualitative way. Section 3.8 briefly describes how the application of refactoring patterns is conducted and Section 3.9 presents concluding remarks and directions for future work.

3.1 Introduction

A software development process is usually composed by a set of interrelated activities, associated to a set of roles, and to the required, produced or modified artefacts. An activity expressed in a process must take place in order to create a piece of software. Examples of activities are: create use cases, analyse risks, model the database, and design the user interface (KRUTCHEN, 2000; COPLIEN; HARRISON, 2005).

A role is an assigned or assumed function or position in a software development process. It defines the responsibilities of the person involved in the process. Examples of typical roles in software development processes are: software architect, project manager, user interface designer, developer, analyst, tester, and technical writers (KRUTCHEN, 2000).

Artefacts are all the documents needed or generated while accomplishing the tasks described in each activity. For example, documents usually manipulated in a software development process are: database models, source code, requirements documents, and software architectural design. Additional elements in a process include guidelines, tool mentors, and roadmaps (KRUTCHEN, 2000).

Related activities are grouped together in disciplines (also called workflows) (TEAM, 2001; KRUTCHEN, 2000). A discipline is an organised set of inter-related activities which encapsulates a core concern of the process. For example, the Rational Unified Process (KRUTCHEN, 2000) has the following set of basic disciplines: business modelling, requirements, analysis and design, implementation, tests, project management, deployment, environment, configuration and change management.

A discipline containing refactoring activities can help the developers to organize the refactoring process in order to minimize the required efforts and maximize the effective results. This is accomplished by improving a particular software module on a set of chosen quality attributes, and by applying a set of selected refactoring patterns that contribute to the improvement of the selected quality attributes on a set of selected elements - chosen by their chances of being improved by the selected patterns.

This chapter has the goal of describing a discipline for refactoring object-oriented and aspect-oriented software. It is comprised of a set of core activities, a set of roles, and a set of artefacts needed to identify, to prioritise refactoring opportunities, and to apply refactoring patterns in software applications.

Figure 3.1 shows these activities using an UML activity diagram.

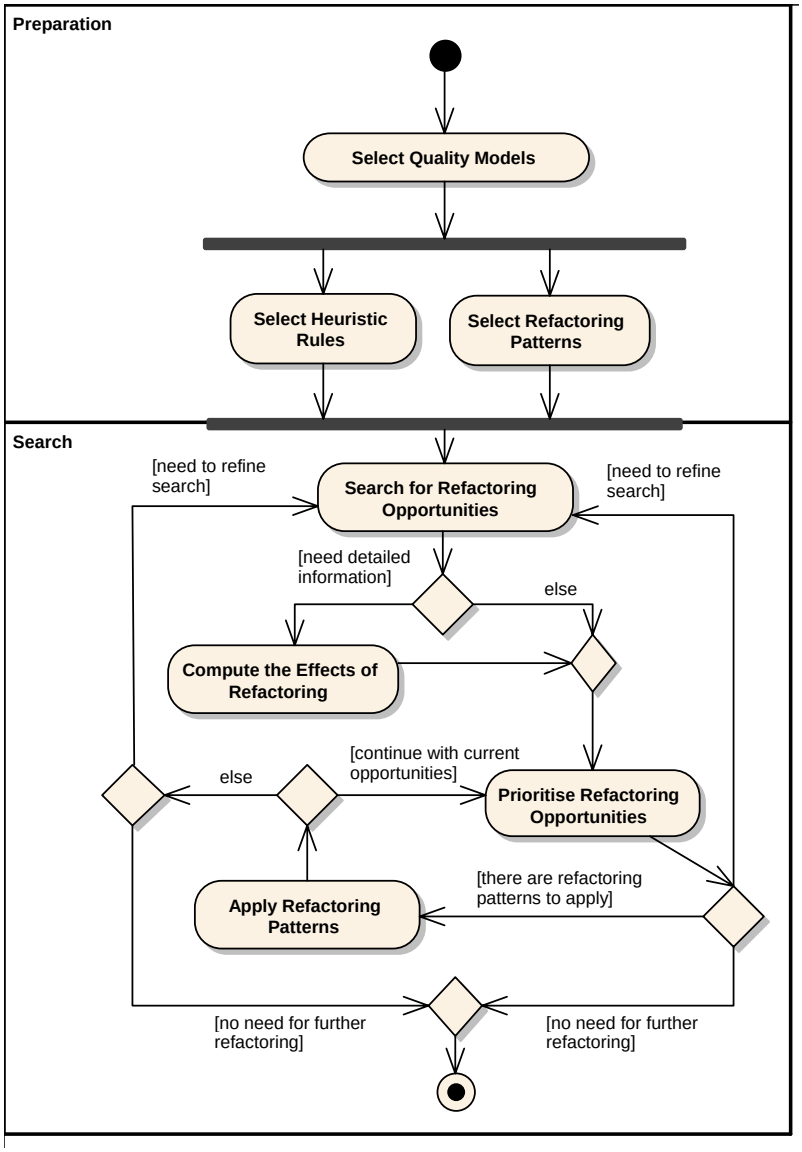

Figure 3.1: Discipline Overview: Main Activities

The activities are briefly described in the following paragraphs, and detailed in further sections:

- **Select or Create Quality Models.** The first step is to select a quality model for which the software application will be evaluated. This activity focus on the defini-

tion of which quality attributes will be considered in the refactoring process. Also, there is the need to define which metrics are associated with these quality attributes, to enable a qualitative assessment of software quality in terms of the selected/created quality model. Examples of quality attributes are: reusability, comprehensibility, legibility, reliability. Examples of metrics are: lines of code, number of operations in module, and depth of inheritance tree.

- **Select or Create Heuristic Rules.** The next step comprises the selection or creation of quantitative functions relating the selected quality attributes and metrics, called heuristic rules. These heuristic rules focus on expressing the relationship between quality models, shortcomings and software elements in a quantitative way. Even though they do not guarantee optimal results, they can provide a good estimative of the qualities the developers are expecting from the software application. Examples of heuristic rules are shown in Chapters 5 and 6.

- **Select Refactoring Patterns.** Parallel to the selection/creation of heuristic rules, the developers must select a set of refactoring patterns from catalogues of refactoring patterns and order them according to their estimated impact on the selected quality model, creating a ranking of refactoring patterns. The focus is on selecting the refactoring patterns that are more likely to improve the software quality attributes. The developer can define thresholds in such ranking of refactoring patterns to increase or decrease the number of evaluated refactoring patterns.

- **Search for Refactoring Opportunities.** Given a set of refactoring patterns, a quality model, a set of heuristic rules, and a set of selected software elements (a package, a set of classes, a single class), the developers can search the software application for possible opportunities to apply those refactoring patterns. The idea is to improve the quality of the software by focusing on the opportunities that are more likely to produce positive effects in a set of selected quality attributes. Examples of refactoring opportunities are shown both in Chapter 5 and 6.

- **Compute the Effects of Refactoring.** After the refactoring opportunities were identified, the developers can then evaluate the effects of the selected refactoring patterns in software artefacts, according to the selected quality model. In this activity, the focus is on a *quantitative* evaluation. The use of impact functions is proposed as a technique to enable the developers to quantitatively compute the effects of refactoring. Another way to compute the effects of refactoring is to apply the refactoring patterns and evaluate the results. Examples of impact functions are shown in Chapter 7.

- **Prioritise Refactoring Opportunities.** After the refactoring opportunities are identified and the effects of each one are computed, the developer evaluates *quantitatively* the proposed changes and decides which ones are advantageous. In this activity, the developer should be able to filter and order the refactoring opportunities and mark them as applicable or not. The developer can then pass to the next activity: *apply refactoring patterns*.

- **Apply Refactoring Patterns.** The next activity is to apply refactoring patterns to the software elements and re-run the available test cases to ensure that the applied refactoring patterns did not break anything. The results can be analysed and, if

needed, the effects of the refactoring patterns can be undone. The developer can restart the search, refining the search criteria, filters, ordering and starting again the analysis of the suggested refactoring opportunities. To make more profound changes in the quality model, in the heuristic rules or in the supported refactoring patterns, the developer can restart the process over from the beginning.

These activities can be divided into two groups: preparation (management) activities and search activities.

Preparation activities include the management of quality models, quality attributes, and metrics, the management of heuristic rules, and the management of refactoring patterns, catalogues of refactoring patterns and rankings of refactoring patterns. These activities precede the search activities. Once the preparation activities are performed, the developers can start to perform search activities.

Section 3.2 describes which are the activities needed for selecting or creating quality models, Section 3.3 describes how to select refactoring patterns and how a ranking of refactoring patterns can be created, and Section 3.4 presents the main activities for selecting heuristic rules (or creating them if needed).

Search activities include the search for occurrences of shortcomings, by computing the values of heuristic rules applied to the software elements selected in a software application. These occurrences of shortcomings, associated with refactoring patterns are the opportunities for applying refactoring patterns. If there is the need to analyse these refactoring opportunities quantitatively, the developers can use impact functions. If not, the refactoring opportunities are evaluated qualitatively and are marked for the application of refactoring patterns or are added to a list of ignored detected refactoring opportunities. Finally, the developers can apply refactoring patterns, run test cases, and if needed, undo the transformations made by the application of the selected refactoring patterns.

The search activities can be performed several times during a software project, during different moments in a software development process. Three main moments are suggested for searching for refactoring opportunities:

- **Refactoring during development.** The developers of classes, when implementing the features described in a use case, a viewpoint (or another way to express requirements), or in a design document can search for refactoring opportunities to apply refactoring patterns during development activities. For example, the developers can find some typical shortcoming in the software elements they are manipulating and apply refactoring patterns, or they can search for refactoring opportunities before committing their changes to a control version system and before deploying the software elements to a testing team or to production. These refactoring activities can focus on quality attributes that contribute to the simplicity, reusability, comprehensibility, and traceability of the software elements, for example.

- **Refactoring before the end of an iteration.** After several deliveries from developers are made and before an iteration ends (in an iterative software development process), the delivered sub-systems can be the focus for refactoring. Note that the set of quality attributes can be different from the ones that the original developers focused. For example, in this moment, shortcomings focusing problem traceability, and error handling can be the focus of refactoring.

- **Refactoring before delivery.** When the system is ready for deployment, there can be an additional refactoring session, focusing on maintenance, and time and space

(performance and size) optimisation, for example.

Section 3.5 presents activities for the definition of the search scope (i.e., which projects, packages, classes, and aspects will be searched for the occurrence of shortcomings). Section 3.6 describes the optional step of evaluating quantitatively the effects of the application of refactoring patterns in the quality of the software elements being evaluated. Section 3.7 describes the activities to analyse the refactoring opportunities and mark which ones will be marked for refactoring and which ones will be ignored. Section 3.8 presents the main activities for the application of refactoring patterns.

3.2 Select or Create Quality Models

Quantitatively, a measure of how *good* is the design of a given class can be represented by the level of satisfaction of the quality attributes the development team wishes to attain. Quality attributes are desired characteristics of software systems (ISO, 2001) and commonly related to non-functional requirements in requirements documents. Common examples of quality attributes, according to Boehm (BOEHM; IN, 1996), are: assurance, interoperability, performance, evolvability, cost and reusability.

The ISO/IEC-9126 (ISO, 2001) is the international standard to deal with product quality and quality models and describes several quality attributes for software systems. There are also quality models for specific domains, such as quality models for component based software development (CBSD) (BERTOA; VALLECILLO, 2002), a model to identify and specify quality attributes that crosscut requirements at an early stage of the software development process (MOREIRA; ARAUJO; BRITO, 2002), quality attributes for web software (OFFUTT, 2002) and quality attributes for the specification of software architectures (KAZMAN; BASS, 1994). Quality attributes are usually described in quality attributes catalogues (BOEHM; IN, 1996; ISO, 2001; MYLOPOULOS; CHUNG; NIXON, 1992).

During software development and evolution, there is the need to use quality models to assess the quality of the software being developed or maintained. Quality models focus on the definition of software quality attributes, which includes the definition of goals, criteria, attributes and metrics. For example, the first recognized models for software quality (BOEHM; BROWN; LIPOW, 1976; MCCALL; RICHARDS; WALTERS, 1977; CAVANO; MCCALL, 1978) incorporate several criteria dealing with product operation, product revision and product transition. These criteria were latter standardized by ISO 9126 (ISO, 2001), which contains six quality goals, each of them having several attributes associated. Basili et al. (BASILI, 1992) propose a method named Goal/Question/Metric (GQM), which includes the definition of a goal, a set of questions and a set of metrics to answer those questions. GQM can be used to create quality models in the approach proposed in this chapter.

The following sections describe the activities and roles needed, the artefacts manipulated and requirements for tool support.

3.2.1 Activities and Roles

The creation or selection of quality models is performed as follows. The quality analyst creates a quality model to assess the quality attributes he wants to evaluate when refactoring. This quality model will be used for the selection of refactoring patterns, for the definition of heuristic rules, for computing the effects of refactoring and for analysing and prioritising refactoring opportunities.

For example, the metric selection can be conducted using the Goal/Question/Metric (BASILI, 1992) method. Other methods or ready-to-use quality models (ISO, 2001) or metric suites (CHIDAMBER; KEMERER, 1994) can be used in this task. If there are already a set of quality models available, the quality analyst can choose which ones will be used for the identification of refactoring opportunities.

The quality analyst is the role responsible for the selection and ranking of refactoring patterns, including the selection of the quality model which will be used to search for refactoring opportunities and also the creation of quality models for the project. Figure 3.2 shows the main activities and artefacts for the selection or creation of quality models.

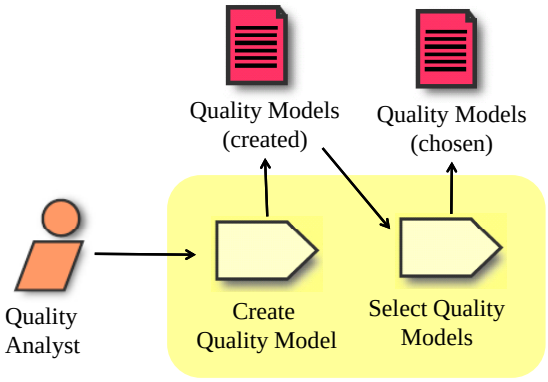

Figure 3.2: Select or Create Quality Models: Roles

3.2.2 Artefacts and Tool Support

Figure 3.3 shows the classes used to represent quality models. The *QualityModel* class defines the abstract representation of a quality model. For example, the developer can choose to use a quality model composed only of quality attributes associated with metrics. Or, he can use a GQM model, comprised of questions, goals and metrics. Each type of quality model can have different structures. Therefore, the *QualityModel* class is specialised to represent different types of organisation for quality models. GQM-based models have associated goals, questions and metrics. Models based on the work of Bohem (BOEHM; IN, 1996) have high level characteristics (which can be seen as high level quality attributes), quality factors (middle level quality attributes) and metrics. ISO 9126 models (ISO, 2001) are organised in hierarchies of quality factors and associated metrics.

A tool supporting these activities should provide mechanisms to:

1. create, update, and remove quality models and its composing elements (quality attributes and metrics);

2. associate metrics with software elements (i.e. the $locc$ metric is associated with classes and aspects);

3. to create, update and remove functions to compute the metrics associated with the quality models.

This activity focuses only on the creation and selection of quality models, comprising quality attributes and metrics. Shortcomings and their respective heuristic rules are defined in the Select or Create Heuristic Rules activity, as well as the computation of the selected metrics of selected software elements.

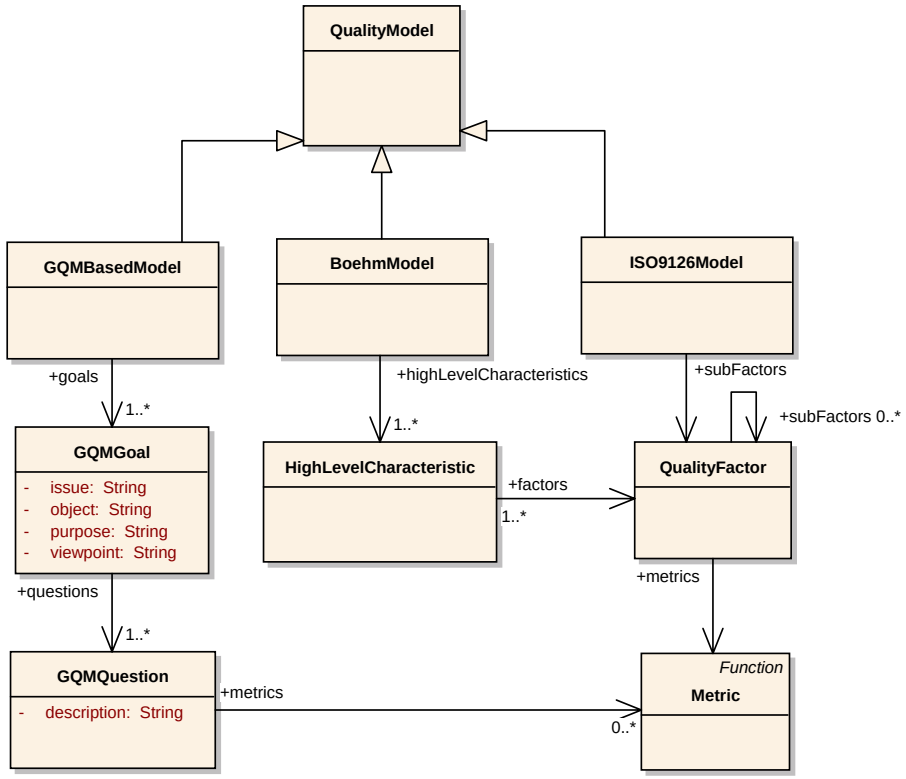

Figure 3.3: Select or Create Quality Models: Artefacts Meta-Model

Existing examples of tools to manage quality models include automated support for the GQM Measurement Process (LAVAZZA, 2000), Logiscope[1], which provides mechanisms to capture metrics measurement and ISO 9126-based quality modelling, and SystemCode[2], which provides indicators to measure the quality of application code, including quality indicators and models for maintainability, changeability and portability of application code. It supports both ISO-9126 quality models as well as customised quality models.

There are also tools focused on the collection of metrics, including CCMetrics[3] for the .NET plataform, Classycle for Java[4], Oink for C++[5], Aopmetrics[6] for AspectJ programs. These tools focus on size, complexity, coupling and cohesion metrics.

The selected quality models is the basis for the selection and creation of heuristic rules, which in turn are used to search for refactoring opportunities, to compute the effects of refactoring on software quality, and to prioritise the refactoring opportunities.

3.3 Select Refactoring Patterns

The developer has to correctly evaluate the trade-offs between refactoring patterns in terms of the affected quality attributes. As these quality attributes can be conflicting with

[1] http://www.telelogic.com/products/logiscope/index.cfm
[2] http://www.metrixware.co.uk/systemcode.php
[3] http://www.riaform.com/utility,ccmetrics,utility.aspx
[4] http://classycle.sourceforge.net/
[5] http://www.cubewano.org/oink
[6] http://aopmetrics.tigris.org/

each other (BOEHM; IN, 1996), the task of selecting optimal refactoring patterns can be hard. Also, as the number of software elements in a software application for which a refactoring pattern is applicable can be high (FOWLER et al., 1999), the developer has to narrow the search for refactoring opportunities to apply refactoring patterns that bring benefits in terms of the expected quality attributes. Current research on the identification of refactoring opportunities (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006) focuses on the improvements of quality attributes considering each individual application of a refactoring pattern, but does not consider how this search can be narrowed to only those refactoring patterns that improve the desired quality attributes of a software system. Currently, there are no, known to the author, automated mechanisms to rank refactoring patterns in terms of quality attributes.

The discipline described in this chapter proposes a set of activities, roles and artefacts to rank refactoring patterns according to a quality model. An example of a ranking method and examples of its application are described in detail in Chapter 4.

3.3.1 Activities

Figure 3.4 shows the main activities to the selection of refactoring patterns (RPs) and the ranking creation.

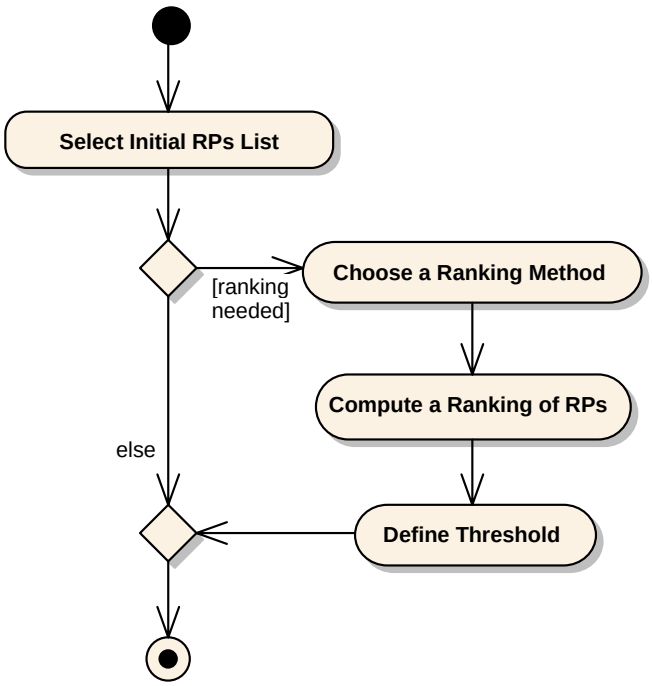

Figure 3.4: Select Refactoring Patterns (RPs): Activities

The following activities are performed:

- **Select Initial List of Refactoring Patterns.** The quality analyst selects an initial set of refactoring patterns that are likely to be used to improve the quality of the software application being developed. He selects a set of refactoring patterns from existing catalogues. There are refactoring catalogues for object-oriented software (FOWLER et al., 1999), aspect-oriented software (MONTEIRO; FERNANDES, 2005a, 2006), design models (MARKOVIC; BAAR, 2005; ZHANG; LIN; GRAY,

2005), and use case models (RUI; REN; BUTLER, 2003; YU; LI; BUTLER, 2004), amongst others. As these catalogues are quite large, it is usually better to focus in sub-set of the refactoring patterns in the catalogues. For example, the Fowler's catalogue comprises around 100 different refactoring patterns.

- **Choose a Ranking Method.** A ranking method is comprised by a set of steps to rank refactoring patterns according to a quality model. This thesis uses AHP to rank a set of refactoring patterns in terms of selected quality attributes. Nevertheless, the quality analyst can choose a different method to rank the refactoring patterns. In this activity, the ranking method is chosen by him.

- **Compute the Ranking of Refactoring Patterns.** The next step is to create a ranking of refactoring patterns using the chosen ranking method. This ranking of refactoring patterns can be used to focus the refactoring effort on the most advantageous refactoring patterns to the software being developed or maintained. In this activity, the quality analyst can use a tool that receives a quality model, a ranking method, a set of refactoring patterns and additional information as input and produces a ranking of refactoring patterns as output.

- **Define Threshold.** After the ranking is created, the developer can define a threshold to reduce the initial set of refactoring patterns to only those that have a ranking value higher than this threshold. He can decrease the threshold value to add more refactoring patterns to the search for refactoring opportunities or increase it if the search for low ranked refactoring patterns is not being fruitful. The use of a threshold narrows the search for refactoring opportunities, focusing on the best ranked refactoring patterns. In this activity, the quality analyst alters the ranking of refactoring patterns, providing a threshold value.

3.3.2 Roles

The following roles are responsible for the selection of refactoring patterns (as shown in Figure 3.5):

- **Developer.** The developer selects a set of refactoring patterns that are available in the IDE or case tool in which he is working or delegates to the quality analyst the refactoring patterns selection.

- **Quality Analyst.** The quality analyst is mainly responsible for creating the ranking of refactoring patterns in terms of the quality model chosen to improve the software through refactoring.

3.3.3 Artefacts

Figure 3.6 shows a model with the structure and the relationship between the artefacts. The artefacts are described as follows.

- **Refactoring Patterns.** Refactoring patterns are described by a name, a context, a solution, a set of mechanisms, and an example. They are usually described in a refactoring catalogue and can be associated with software elements, impact functions and heuristic rules. One possible way to associate refactoring patterns with software elements with a low coupling is shown in Figure 8.2. Refactoring patterns are associated with impact functions through the arguments of such functions.

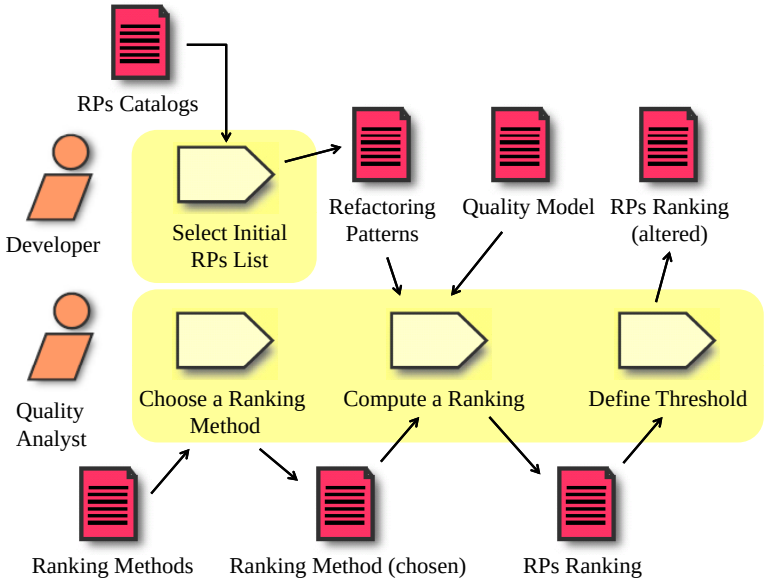

Figure 3.5: Select Refactoring Patterns: Roles

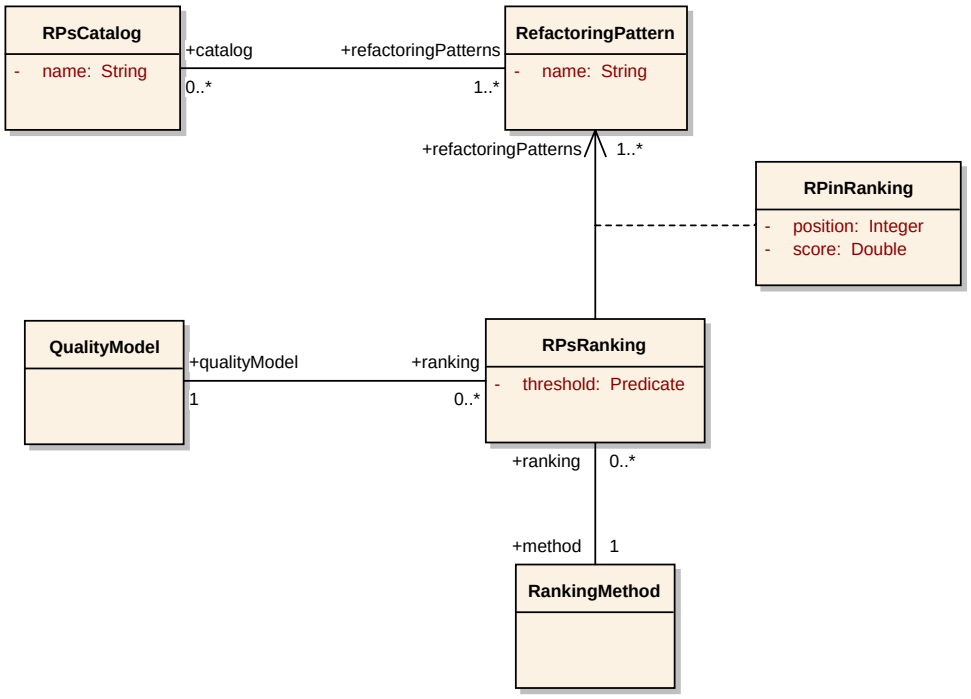

Figure 3.6: Select Refactoring Patterns (RPs): Artefacts Meta-Model

Figure 3.15 shows that the arguments are software elements, which are in turn associated with a refactoring pattern through refactoring opportunities.

- **Refactoring Catalogue.** The catalogue of refactoring patterns is used by the developer or the quality analyst to select the refactoring patterns to be ranked or selected to be used in the remaining activities.

- **Ranking Method.** A ranking method is used to order the refactoring patterns in a

way that the refactoring patterns that are more likely to improve the quality model are the best ranked patterns.

- **Ranking of Refactoring Patterns.** This artefact lists the refactoring patterns together with their position in the ranking and a score that can represent how much one refactoring pattern is more advantageous than the others, in terms of a set of quality attributes. This score can be computed using AHP and can be used by the developer to decide which refactoring patterns to include in the search and which not. For example, the developer can choose to only search for refactoring patterns that have scores higher than an arbitrary value. Or, he can search for the top n refactoring patterns in the ranking, where n is an arbitrary number. The ranking can also maintain references to the ranking method used and the quality model used to rank the refactoring patterns.

A tool supporting these activities should provide (i) integration with refactoring patterns catalogues and (ii) integration with ranking method tools. It should be possible to:

- Create, read, update and delete refactoring patterns;
- Create, read, update and delete quality models;

3.4 Select or Create Heuristic Rules

A heuristic (RUSSELL; NORVIG, 2002) is defined as any technique that improves the average-case performance on a problem-solving task and is usually an approximation of knowledge. In this thesis, a heuristic is expressed as a rule to quantitatively evaluate a set of quality attributes of a given element of a model. Heuristic rules can help to to evaluate the level of satisfaction of quality attributes in a given software application and to spot opportunities to improve software applications and their quality.

3.4.1 Activities and Roles

The following activities are needed to be performed by the developer or by a quality analyst. Figure 3.7 shows such activities.

- **Select Heuristic Rules.** In this activity, the developer or the quality analyst can select a set of heuristic rules associated with a quality model retrieved from a knowledge base.
- **Create Heuristic rules.** To create a heuristic rule, the developer can start with an aggregation, such as: $f = \sum_{i=1}^{n} w_i * fqa_i$, in which $fqa_1 \ldots fqa_n$ are functions that measure the quality attributes $qa_1 \ldots qa_n$. The $w_1 \ldots w_n$ elements of the function are the weights of each quality attribute in the heuristic rule. As a development team is usually concerned with several quality attributes simultaneously, a multi-criteria approach can be used to take all the quality attributes into account.

A quality analyst is responsible to create the heuristic rules that will be used in the refactoring activities. If, for example, he wants to create a heuristic rule based on quality attributes and metrics, he needs to first select the quality attributes, then select a set of metrics (using a metric selection process, such as GQM) to create a function to quantitatively evaluate the quality attributes using a set of metrics.

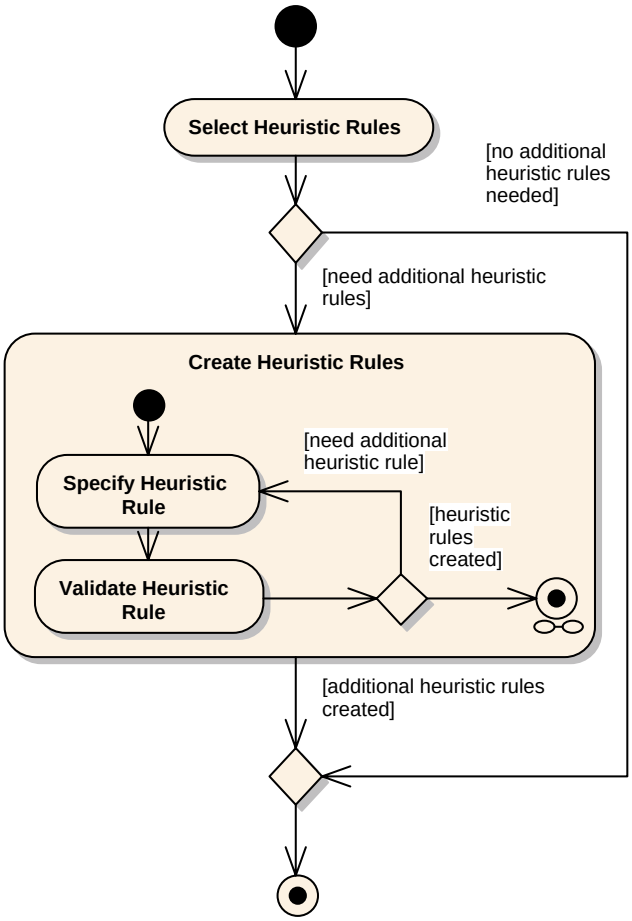

Figure 3.7: Select or Create Heuristic Rules: Activities

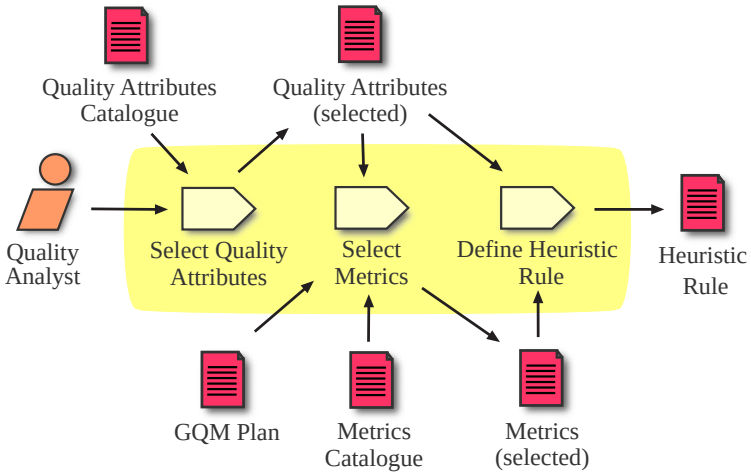

Figure 3.8: Select or Create Heuristic Rules: Roles

3.4.2 Artefacts

This section discusses a conceptual model describing the main concepts associated with the definition of heuristic rules. Figure 3.8 shows such model.

- **SoftwareElement.** In this case, a software element is any part of an artefact. For

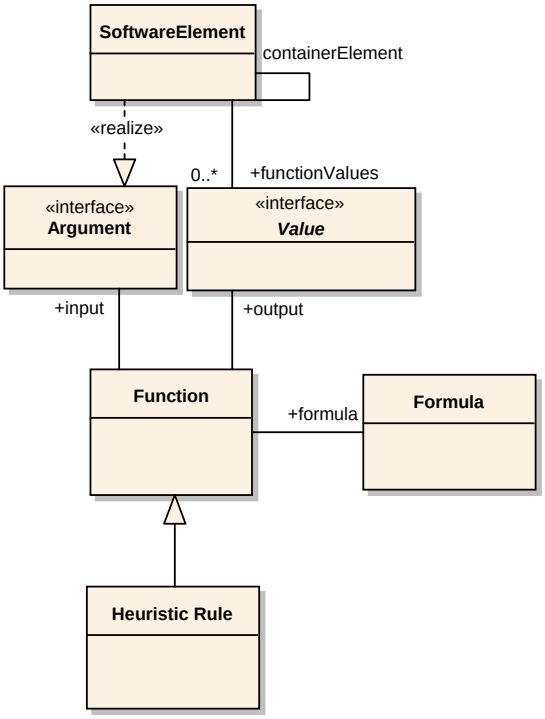

Figure 3.9: Create Heuristic Rules: Artefacts

example, in object-oriented code, a software element can be a class, a method, and an attribute. These software elements are passed to the heuristic rules as needed.

- **Function.** This class represents any mathematical function. Mathematical functions are composed by a set of arguments (also called parameters), a value, which is returned by the computation of the function, and a formula, which specifies how the parameters are used and how the value is computed.

- **Argument.** This interface represents the parameters passed to a function. For example, if the function computes the $locc$ metric, the argument can be an aspect or a class. If it is an impact function to compute the impact of the *Move Method* refactoring pattern, the arguments can be the evaluated metric, the method being moved and one additional argument representing the destination class or aspect.

- **Value.** This interface represents the values returned by a function. If the function computes the nom metric, for example, the value is the number of methods of the class or aspect passed as argument to that function.

- **Formula.** The formula of a function is the specification of how a function is computed. It is the right side of a mathematical function. For example, the formula of the defined to compute the nom metric is defined, in Chapter 9, as: $|\mathcal{M}| + |\mathcal{A}| + |\mathcal{MD}| + |\mathcal{CD}|$.

- **Heuristic Rule.** A heuristic rule is a function created to evaluate quantitatively one or more software elements. For example, the heuristic rule defined in Chapter 6, is defined as: $f(x) = 0.4 * simplicity(x) + 0.6 * reusability(x)$, where simplicity and reusability are other functions.

Tools must include support to:

- Create, read, update, and delete shortcomings;
- Create, read, update, and delete metrics;
- Create, read, update, and delete heuristic rules;
- Associate shortcomings with heuristic rules;
- Compute the values of metrics;
- Compute the values of heuristic rules;

3.5 Search for Refactoring Opportunities

In software applications, several opportunities for applying refactoring patterns can be found. The difficulty is to determine which of these opportunities can improve the qualities the developers wish to satisfy. The following questions are presented in such context:

- To what software elements refactoring patterns should be applied?
- Which refactoring patterns are applicable?
- What is gained by the application of a refactoring pattern to a given software element?

A software element can be the target for several different refactoring patterns. Using a set of heuristic rules, the developer can (i) search for the refactoring opportunities in the software elements, and (ii) check if the refactoring patterns for each target element are applicable. A refactoring pattern is applicable to a software element if all the preconditions of the refactoring pattern are satisfied.

This section describes the main activities, roles and artefacts needed to search for refactoring opportunities in existing software using heuristic rules and impact functions.

3.5.1 Activities

Figure 3.10 shows the main sub-activities for the Search for Refactoring Opportunities activity.

- **Define Scope.** The scope of a search is defined as the projects, packages, and software elements that will be used in the search for refactoring opportunities.
- **Define Levels of Successive Refactoring.** The search for refactoring opportunities can be extended to sequences of two or more refactoring patterns, as described in detail in Chapter 8.
- **Compute Elements in the Scope.** In this step, the software elements which can be a target for refactoring are computed, according to the scope defined in the previous steps.

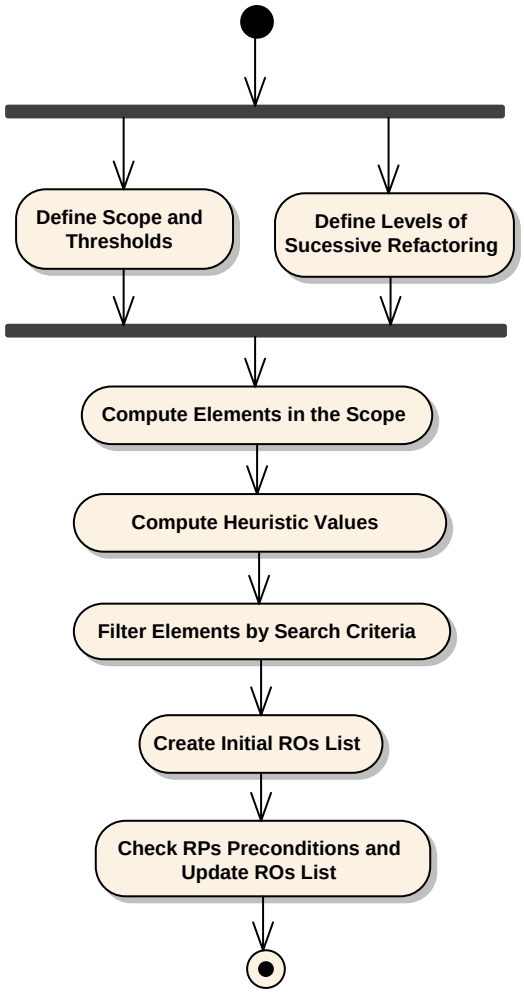

Figure 3.10: Search for Refactoring Opportunities (ROs): Activities

- **Compute Heuristic Rules.** The first step in finding refactoring opportunities is to compute the metric values of the software elements and then compute the values of the heuristic rules.

- **Create an Initial Refactoring Opportunities List.** After the software elements in the scope and the values of the heuristic rules are computed, an initial list of refactoring opportunities is created. This list will be used to prioritise the application of refactoring patterns.

- **Check Refactoring Patterns Pre-conditions and Update Refactoring Opportunities List.** The last step is to check the pre-conditions of the refactoring patterns and update the refactoring opportunities list. This step can be postponed if there are too many refactoring opportunities (the cost of computing the pre-conditions can be estimated based on previous computations).

3.5.2 Roles

The search for refactoring opportunities activities can be all performed by the developers with the help of automated tools. The developers start by setting the configurations needed for the search, including the scope (which projects, packages, and classes), and the

levels of successive refactoring (as defined in Chapter 8). With the configuration in place, the next step is to compute which elements are on the scope and filter those elements by the conditions described in the scope configuration.

With all the software elements within the defined scope, and the previously selected refactoring patterns, the developer can create an initial list of refactoring opportunities, including all the cases in which the refactoring patterns can be applied to the software elements in the scope. The last step is to check if the preconditions of each refactoring pattern application is satisfied (this checking is usually expensive in terms of time, so this step can be delayed until the actual application of the refactoring patterns). Figure 3.11 shows the activities and artefacts involved.

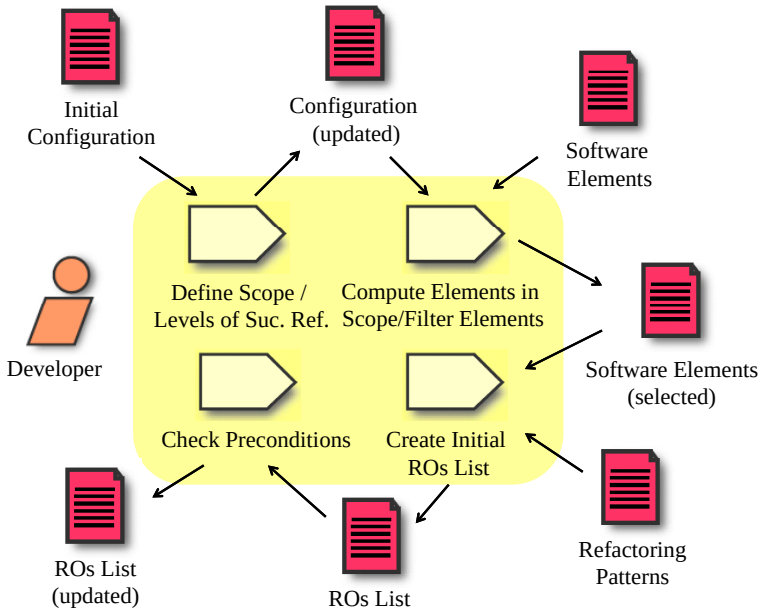

Figure 3.11: Search for Refactoring Opportunities (ROs): Roles

3.5.3 Artefacts

This section shows a conceptual model describing the main concepts associated with the definition of heuristic rules, impact functions and search for refactoring opportunities.

Figure 3.12 shows the main classes and its attributes and relationships, described as follows:

- **Software Element.** A software element can be any element in a software artefact, such as a class, an attribute, a relationship, and a use case.

- **Metric.** A metric is described by a name and is implemented as a function. Functions have arguments, output values and a formula, as shown in Figure 3.9. A framework that implements the search for refactoring opportunities must provide integration with a metric collector software application.

- **Value.** The instances of this class contain a value for a given heuristic rule. For example, when the heuristic rule is computed, its output can be a boolean value (the software element has the *anonymous pointcut* shortcoming for example), a numeric value (the reusability of the class is of 0.53, for example) or other kind of value.

- **Shortcoming.** A shortcoming is a deficiency, inadequacy or incompleteness that a software element can have. Typical shortcomings are described by a name, applicability, a set of examples and a set of heuristic rules, to automatically detect their occurrence.

- **Refactoring Opportunity.** A refactoring opportunity is composed by a set of selected software elements, a refactoring pattern, and a heuristic rule (which is associated with a shortcoming).

- **Refactoring Pattern.** A refactoring pattern is a behavioural preserving transformation, which is used to improve the quality of a software application being developed. It is comprised by a name, a motivation, a set of parameters and possibly other basic attributes. The application of a given refactoring pattern must satisfy a set of pre and post conditions. The pre-conditions specify when it is possible to apply the refactoring pattern. The post conditions express the changes that must occur whenever the refactoring pattern is applied.

- **List of Refactoring Opportunities.** After the search is conducted, a list of refactoring opportunities is created. This list is evaluated to decide which of the refactoring patterns associated with the refactoring opportunities will be applied.

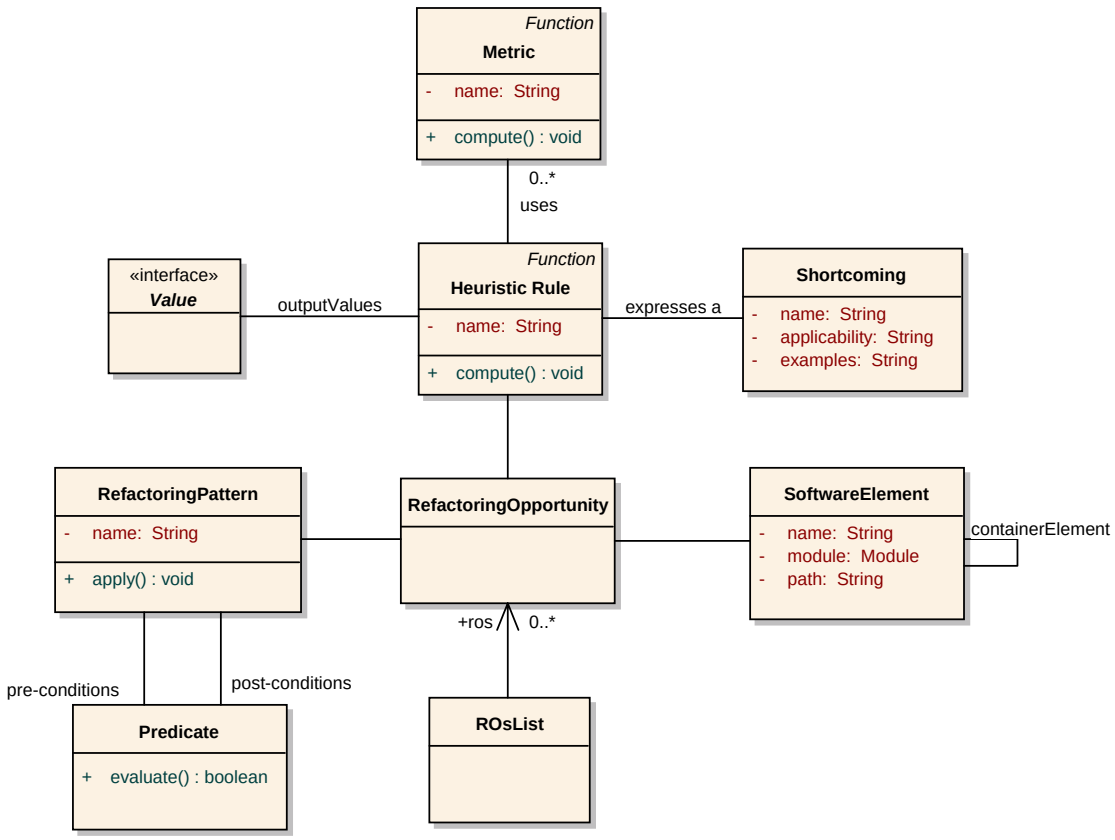

Figure 3.12: Search for Refactoring Opportunities: Artefacts Meta-Model

3.6 Compute the Effects of Refactoring

Several refactoring patterns have been proposed to improve the design of existing software (OPDYKE, 1992; FOWLER et al., 1999; GARCIA et al., 2004; MONTEIRO; FERNANDES, 2004, 2005a, 2006). However, the benefits of applying each refactoring pattern are context dependent. For example, consider the *Extract Method* and the *Inline Method* refactoring patterns. The *Extract Method* refactoring pattern is usually applied when a method is too long, the sentences are not in the same level of abstraction, or it is clearer to have a new method encapsulating a particular set of sentences. For example, if the developer applies *Extract Method* too frequently, he will have a system with too many methods and too much indirection. In this case, the *Inline Method* refactoring pattern can be used to get rid of methods that are not doing too much and are not useful enough to exist as a separated method.

Another issue that can affect when a particular refactoring pattern can bring benefits is regarding the software quality attributes, which can be a different set for every project. Consider, for example, conflicting quality attributes, such as performance vs. security, simplicity vs. flexibility, optimization vs. legibility. Each refactoring pattern affects differently each quality attribute, so the selection of which refactoring patterns to select depends on the relative importance of the quality attributes required for the project.

Impact functions (BOIS; MENS, 2003; BOIS, 2006) are used to evaluate the impact of a refactoring pattern on software metrics and describe the changes in the metric values when a given refactoring pattern is applied. Any time a developer finds a refactoring opportunity and is not sure about the implications of the transformation the associated refactoring pattern will cause, an impact function can help him to assess the effects of its application. Using the impact functions, the developer chooses proper refactoring patterns according to his needs.

The decision of moving features between software elements can be supported by the impact functions, which help the developer to evaluate the metric values of the application of each refactoring pattern. The use of these functions can show which candidates for the application of refactoring patterns have more impact in terms of metric values. Furthermore, when there are many refactoring opportunities, the developer has to focus on those that provide more improvements in the software being developed.

3.6.1 Activities

Figure 3.13 shows the main activities for the Compute the Effects of Refactoring activity.

The activities are described as follows.

- **Retrieve Impact Functions for the Selected Refactoring Patterns and Metrics Used.** In this activity, the impact functions regarding the selected refactoring patterns and the selected heuristic rules are retrieved from a knowledge base. If no suitable impact functions are found, a quality analyst can create new ones.

- **Create Impact Functions.** This activity encompasses the creation of functions to predict the values of heuristic rules or metrics without the need to apply the refactoring patterns. Whenever a developer comes upon an opportunity for refactoring and is not sure about the implications of the transformation, an impact function helps to assess the effect of the refactoring.

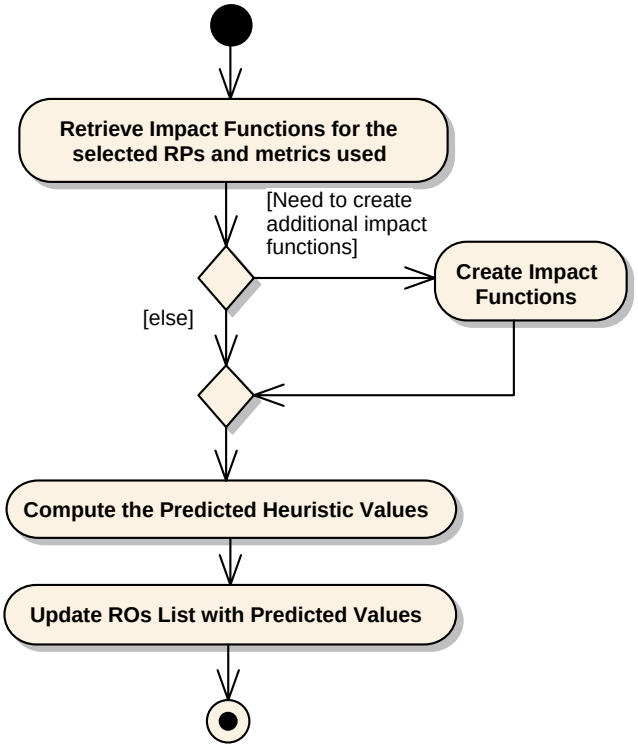

Figure 3.13: Compute the Effects of Refactoring: Activities

- **Compute the Predicted Values of Heuristic Rules.** In this activity, the predicted values of the heuristic rules are computed and associated with the corresponding software elements.

- **Update Refactoring Opportunities List.** The refactoring opportunities list is updated to contain the new predicted values. The developer can then evaluate the changes in the predicted values if he applies each one of the selected refactoring patterns.

3.6.2 Roles

The following roles are responsible for the Compute the Effects of Refactoring subactivities.

- **Quality Analyst.** The quality analyst is responsible for retrieving the impact functions for the selected heuristic rules and refactoring patterns. He is also responsible for creating new impact functions if there are no impact functions available for all the heuristic rule vs. refactoring pattern pair. The retrieval can be provided by a tool.

- **Developer.** The developer computes the predicted values of the heuristic rules and updates the refactoring opportunities list with those values. Both activities can be automated.

Figure 3.14 shows the relationship between roles, artefacts and activities for computing the effects of refactoring on software quality.

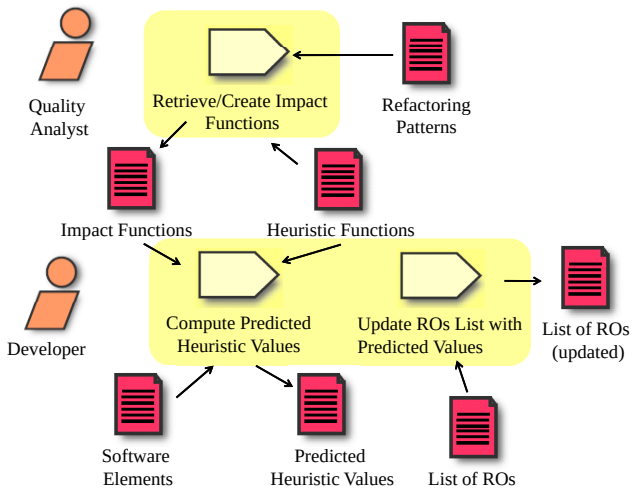

Figure 3.14: Compute the Effects of Refactoring: Roles

3.6.3 Artefacts

The artefacts needed for and produced by the Compute the Effects of Refactoring activity are described as follows.

- **Selected Heuristic Rules.** This artefact contains the heuristic rules selected for the evaluation of the software artefacts. These heuristic rules are used to create or select impact functions to compute the effects of refactoring in a software application.

- **Selected Refactoring Patterns.** This is a list of the refactoring patterns selected by the developer as an input for the search for refactoring opportunities.

- **Impact Functions.** Impact functions are created for each heuristic rule/refactoring pattern pair (or for each metric/refactoring pattern pair). These functions compute the predicted value of the heuristic rule if the refactoring pattern is applied to a given software element.

- **Software Elements.** Software elements are the building blocks of software: requirements, use cases, classes, methods, and variables. These elements comprise all the elements that can be affected by the application of a refactoring pattern.

- **Predicted Values of Heuristic Rules.** These are the values of the computation of the heuristic rules together with the impact functions. Software elements can have several values associated for the heuristic rules and predicted values of the heuristic rules. While values of heuristic rules are computed with the heuristic rules, the predicted values of a heuristic rule are computed with the help of impact functions.

- **List of Refactoring Opportunities.** These opportunities associate software elements to refactoring patterns and are used to store the predicted values of a heuristic rule.

Figure 3.15 shows the main classes representing the artefacts, their properties and relationships.

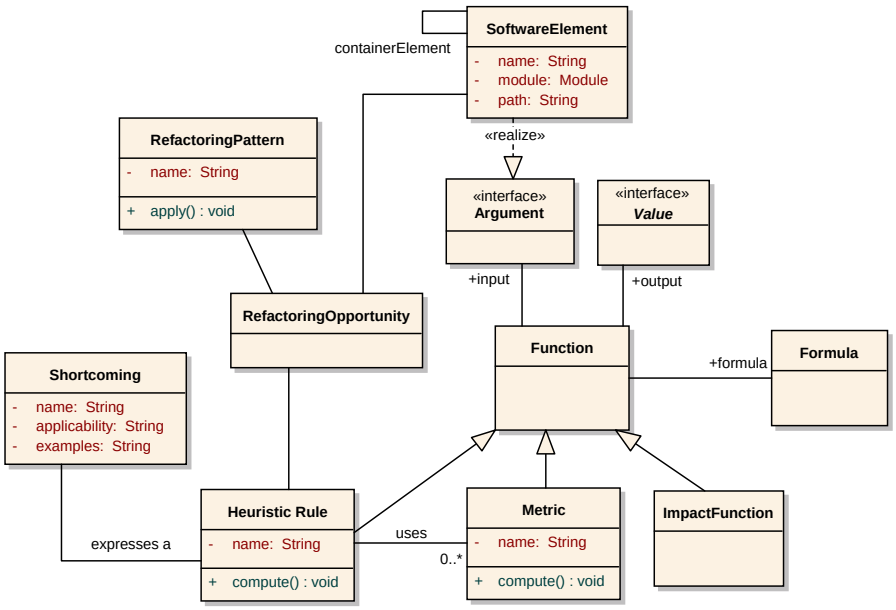

Figure 3.15: Compute the Effects of Refactoring: Artefacts Meta-Model

Tool support for using impact functions is desirable to provide mechanisms to compute metric values without the need of applying refactoring patterns in a practical way. The developer can use the existent impact functions associated with a set of refactoring patterns or can create new ones.

Each refactoring pattern has a set of participants (source aspect and destination aspect, for example) and can be associated with a set of metrics. For each pair $\mathcal{P}_m$ = (participant, metric), the developer can inform a different impact function. The developer can also associate more than one metric and impact function in a single operation.

The classes representing refactoring patterns are instantiated and the impact functions can be computed for concrete participants, provided by the user. This computation returns the modified metric values. Refactoring in sequence can be performed by sequentially computing individual impact functions.

Tools must support mechanisms to:

- Create, read, update, and remove impact functions;
- Compute metrics;
- Compute impact functions;
- Integrate the computation of impact functions with refactoring tools (i.e. to show the impact of refactoring patterns within a particular IDE).

Future work can focus on expanding the existing refactoring patterns and impact functions, providing integration with metric collectors for object-oriented and aspect-oriented software, commercial IDEs and refactoring tools.

3.7 Prioritise Refactoring Opportunities

After searching for the applicable refactoring patterns (or sequences of patterns) and computing the effects of these applications in software quality, the developer has to choose

which of the suggested refactoring opportunities are really advantageous.

In this activity, the developer manipulates the refactoring opportunities by means of ordering and filtering. When one of the refactoring opportunities seems suitable, he can then analyse it more carefully, to understand its effects on the artefacts being manipulated.

The developer then can choose to discard or to mark the opportunities for refactoring. The following sections briefly describe the main activities, roles and artefacts needed for this activity.

3.7.1 Activities and Roles

Figure 3.16 shows the main flow of activities when analysing a set of refactoring opportunities.

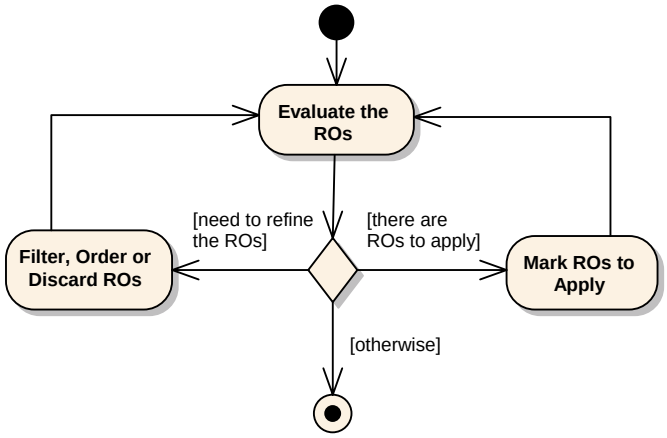

Figure 3.16: Prioritise Refactoring Opportunities (ROs): Activities

The main activities are described as follows.

- **Evaluate the Refactoring Opportunities.** In this activity, the developer evaluates the best ranked refactoring opportunities and chooses those that he finds suitable to the current development context.

- **Filter, Order or Discard Refactoring Opportunities.** If there is the need to refine the search for refactoring opportunities, the developer changes the filters, orders the refactoring opportunities or discards those refactoring opportunities that are not appropriate. After the set of refactoring opportunities is refined, the developer can continue to evaluate them.

- **Mark Refactoring Opportunities to Apply.** Those refactoring opportunities which the developer finds suitable to apply the associated refactoring pattern are marked for refactoring. Later, the developer can provide additional parameters to the refactoring patterns and apply them.

The only role responsible for the activities of refactoring opportunities evaluation, analysis of trade-offs and prioritisation is the developer. Figure 3.17 shows the relationship of the developer with the main activities and artefacts.

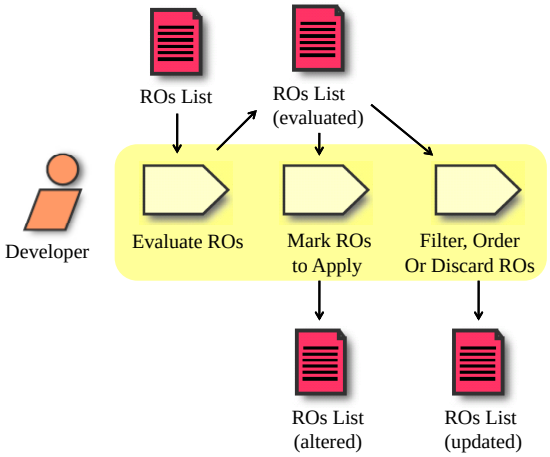

Figure 3.17: Prioritise Refactoring Opportunities (ROs): Roles

3.7.2 Artefacts

The only artefact manipulated is the list of refactoring opportunities, which is first evaluated and then can be further refined or have opportunities marked for refactoring. The main classes for this activity are shown in Figure 3.18.

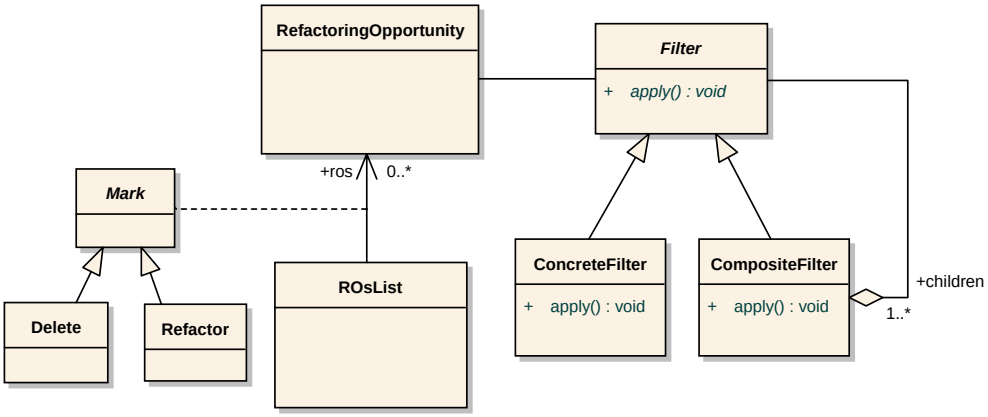

Figure 3.18: Prioritise Refactoring Opportunities: Artefacts Meta-Model

Tool support for this activity should have the following features:

- The capability to select and mark opportunities for later refactoring;
- The capability to hide or delete refactoring opportunities from the List of Refactoring Opportunities;
- The capability to order and filter refactoring opportunities by metric values, values of heuristic rules, predicted values of heuristic rules, the difference between the predicted values and the values of heuristic rules, by package, by class, and by refactoring pattern.

3.8 Apply Refactoring Patterns

The goal of refactoring is to improve the quality of the artefact itself and the overall software quality. This improvement can be evaluated qualitatively or quantitatively. In this activity the focus is to apply the suggested/marked refactoring opportunities and to test the software application to evaluate if there are no side-effects of the application of the refactoring patterns associated with the refactoring opportunities. The developer needs, for some refactoring patterns, to provide concrete parameters to the refactoring pattern being applied. For example, if an Extract Interface refactoring pattern is suggested, the developer has to provide the name of the new interface.

This section describes the main activities, roles and artefacts to apply refactoring patterns to software elements after refactoring opportunities were identified, filtered, ranked, analysed and marked for refactoring.

3.8.1 Activities

Figure 3.19 shows the main flow of activities when refactoring a set of software elements marked for refactoring.

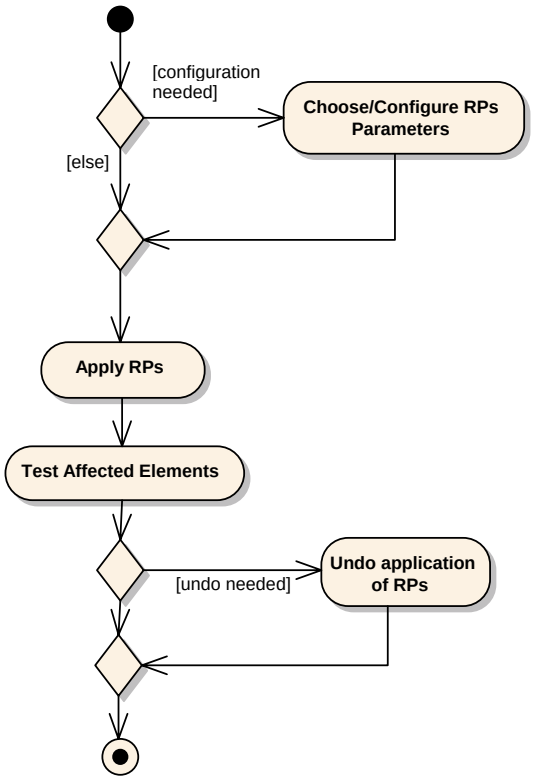

Figure 3.19: Apply Refactoring Patterns (RPs): Activities

The following activities are needed to apply the refactoring patterns associated with the refactoring opportunities.

- **Choose/Configure Parameters.** Some refactoring patterns need parameters, so the developer can choose the best value for it. It can include naming issues (for classes, new methods, and renamed variables) or destination classes, for example (for which super-class the developer should pull up a method).

- **Apply Refactoring Patterns.** The developer applies the marked refactoring patterns in the software elements. The refactoring tool then performs the transformations and reports to the user any errors which occur in this process. This activity is usually automated.

- **Test Affected Elements.** The software elements are then tested, using test cases (unit tests, and integration tests) to check if the all transformations occurred successfully. If errors are reported, the developer can choose to undo the changes or analyse both the software elements and the test cases to see which are the issues that are causing the errors.

3.8.2 Roles

The primary role responsible for this activity is the same role responsible for the application of refactoring patterns to the software elements. For example, if source code is being a target for refactoring, the developer is responsible for this activity. If models are being transformed by refactoring patterns, the analyst or the designer is the responsible role. Figure 3.20 shows the main artefacts associated with this role and activities.

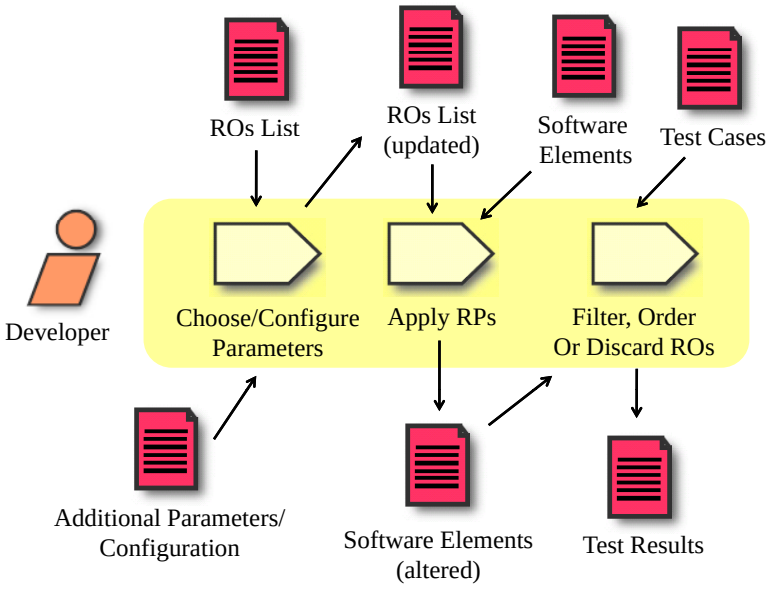

Figure 3.20: Apply Refactoring Patterns: Roles

3.8.3 Artefacts

The following classes can be used to represent the artefacts needed for applying refactoring patterns to software elements.

- **Refactoring Opportunity.** Refactoring opportunities are represented as an association of a refactoring pattern with one or more software elements and with a heuristic rule. A list of such objects is used to drive the application of refactoring patterns.

- **Refactoring Pattern.** Each refactoring opportunity has a refactoring pattern associated with it. There is the need of supporting the refactoring pattern in the refactoring tool.

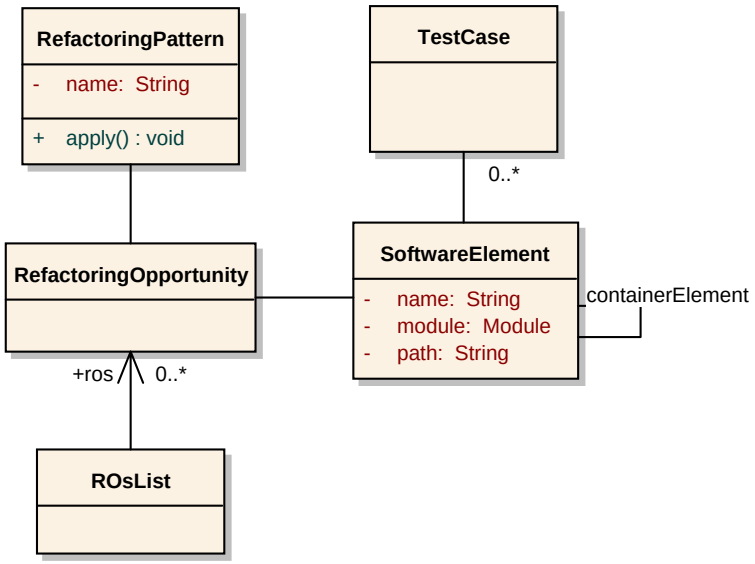

Figure 3.21: Apply Refactoring Patterns: Meta-Model

- **Refactoring Opportunities List.** These are concrete lists of refactoring opportunities, which were created in the previous activities.
- **Software Element.** Each refactoring opportunity has one or more software elements associated. These elements will be transformed by the application of the refactoring patterns in the list.
- **Test Case.** Test cases are used to evaluate if the application of the selected refactoring patterns in the associated software elements do not change the behaviour of the application.

Tool support is highly desirable for the refactoring activities and is usually present in modern IDEs and Case tools. For example, the Eclipse IDE (`http://www.eclipse.org`) and the IDEA IDE (`http://www.jetbrains.com/idea/`) provide more than 50 different refactoring patterns for manipulation of classes, methods, fields, interfaces, and statements. Case Tools, such as the Rational Rose and Rational Software Architect (`www.ibm.com/software/rational/`), provide such capabilities as well, with the advantage of synchronizing design models and source code (if one of them changes, the other one is also changed).

3.9 Conclusions

There are several activities involved in a refactoring process considering improvements on the software quality. First, there is the need to select or create a suitable quality model to drive the entire process. Then, considering the selected quality model, the developer can then select the refactoring patterns that are more likely to produce positive effects in the artefacts being the target for refactoring patterns. Then, heuristic rules can be created to quantitatively evaluate the software elements.

Having created the heuristic rules, the developer can then search for refactoring opportunities, and evaluate the effects of the refactoring patterns associated with those op-

portunities using impact functions. The next step is to analyse the proposed refactoring opportunities by filtering, ordering and marking refactoring opportunities for the application of the associated refactoring pattern. Typically, after a set of refactoring opportunities are marked, the developer can apply the refactoring patterns to the elements associated with the refactoring opportunities.

Future work can focus on the validation of such discipline, including the use of the proposed activities in real world projects. This validation can help to assess the advantages and disadvantages of the proposed approaches, and also the obtained gain with full tool support.

4 RANKING REFACTORING PATTERNS WITH THE ANALYTICAL HIERARCHY PROCESS

This chapter proposes an approach to rank refactoring patterns in terms of a set of quality attributes. It is organized as follows. Section 4.1 describes the main motivation for ranking refactoring patterns and describes the goals of this chapter. Section 4.2 shows how to rank refactoring patterns according to a set of quality attributes using AHP, while Section 4.3 describes a case study. Section 4.4 describes how to improve the pairwise comparisons and ranking accuracy. Section 4.5 describes the tool support developed to automate the main activities of the approach. Section 4.6 describes related work and Section 4.7 concludes the chapter.

4.1 Introduction

The evaluation of all the possible refactoring opportunities is costly. There are several applicable refactoring patterns, there are several software elements to be evaluated, there are several metrics whose values must be computed and analysed, and there are several sequences to choose from. To reduce the number of refactoring opportunities to be evaluated, the developer can focus on reducing the number of refactoring patterns, on reducing the scope (by reducing the software elements), on reducing the set of typical shortcomings, on selecting a set of quality attributes, or on reducing the number of sequences to be evaluated.

One of these ways of reducing the number of refactoring opportunities to be evaluated is to reduce the number of refactoring patterns. The developer selects refactoring patterns with the potential of bringing advantages in terms of chosen quality attributes to the current project in general. This is the focus of this chapter and can be addressed by creating a ranking of refactoring patterns that improve a set of quality attributes of software application.

Strategies for reducing the scope by selecting specific software elements, by selecting the software elements according to a measure of quality, and by selecting shortcomings for which the developer will search for refactoring opportunities is the focus of Chapters 5, 6 and 3.

Another strategy for reducing the number of refactoring opportunities deals with prioritising them by *evaluating the effects* of each application of the selected refactoring patterns (individually or in sequences), as each refactoring pattern can have different effects depending on the software elements it manipulates. This strategy is outside the scope of this chapter and is discussed in more details in Chapters 7 and 8, respectively.

For the selection and ranking of refactoring patterns according to a set of chosen qual-

ity attributes, the developer can use one of several multi-criteria decision methods. Such methods offer the possibility to find, given a set of alternatives and a set of decision criteria, the best alternative. To solve MCDM problems many techniques have been proposed [25]: direct scoring and ranking methods, trade-off schemes, distance-based methods, value and utility functions, and interactive methods. The Analytical Hierarchical Process (AHP) (SAATY, 1990, 2003) was the method selected in this thesis to rank refactoring patterns according to a quality model.

The Analytical Hierarchical Process (AHP) (SAATY, 1990, 2003) was selected for the following reasons:

- It allows pairwise comparisons (a kind of tradeoff and interactive method), which seems appropriate to handle the kind of problems in hand. The relative importance of quality attributes essentially depends on the requirements of each project and it is not easily measured quantitatively.

- These pairwise comparisons, besides being used for the ranking of required quality attributes of a given project, can also be used to express the relative importance of one refactoring pattern over other refactoring patterns. The advantage of using AHP is that the developers can state qualitatively which refactoring pattern is better than another in terms of a given quality attribute. This qualitative information is mapped to quantitative values using AHP, and the pairwise comparisons can be automatically computed with a small effort.

- It provides a simple aggregation process (BRITO et al., 2007). There are no special skills to use the method.

- It helps guarantee the logical consistency of many human-based judgements, as well as synthesizing a wide-range of data in a single solution (BRITO et al., 2007).

- It has been applied successfully in diverse domains, such as environmental assessment (GELDERMANN; SPENGLER; RENTZ, 2000), land management (JOERIN; MUSY, 2000), maintenance strategy (BEVILACQUA; BRAGLIA, 2000) and construction partnering process (CHENG; LI, 2002). In the context of software development, it is being employed to optimise the value and cost in requirement analysis (JUNG, 1998), to the decomposition of interdependent task group for concurrent engineering (CHEN; LIN, 2003) and to define optimisation models for quality and cost of modular software systems (JUNG; CHOI, 1999), for example.

This chapter proposes an approach to rank refactoring patterns in terms of a set of quality attributes. The Analytical Hierarchy Process (AHP) multi-criteria decision method is used to express:

- How much one quality attribute is more important than the other quality attributes (this is called *relative importance* (SAATY, 1990));

- How much one refactoring pattern is more important than other refactoring patterns in regards to each quality attribute.

The main tasks are the:

- The creation of pairwise comparisons for quality attributes;

- The creation of pairwise comparisons for refactoring patterns;
- The computation of the following rankings:
  - Ranking of quality attributes;
  - Rankings of refactoring patterns for each quality attribute;
  - The overall ranking of refactoring patterns regarding all the (ranked) quality attributes.

The creation of a ranking of refactoring patterns for a hypothetical software project is exemplified using three quality attributes and four refactoring patterns. Although the example is based on refactoring patterns for object-oriented software, the approach can be used in other paradigms.

4.2 Creating a Ranking with AHP

This section describes the main steps for ranking refactoring patterns according to a set of preferred quality attributes using AHP.

Starting with a set of refactoring patterns and a set of selected quality attributes, the developer performs the following steps to generate a ranking of refactoring patterns according the preferred quality attributes:

1. *Create pairwise comparisons for quality attributes:* The first step is to create pairwise comparisons between the selected quality attributes. This is the developer's main task in this approach, as the relationship of quality attributes is usually specific to a project.

2. *Create pairwise comparisons for refactoring patterns:* A tool provider can make available a knowledge base containing pairwise comparisons for typical refactoring patterns and quality attributes. The developer can then retrieve the pairwise comparisons from this base or add new ones to the knowledge base (if there are no pairwise comparisons available for a particular refactoring pattern or quality attribute).

3. *Compute the quality attributes ranking:* In this task, a pairwise matrix is created to express the relationship between the quality attributes. The quality attributes ranking is the eigenvector of the quality attributes pairwise matrix. This step and the next ones can be automated.

4. *Compute the rankings of refactoring patterns versus quality attributes:* For each quality attribute, a pairwise matrix is created to quantitatively express the relationship of the refactoring patterns vs. the quality attribute. The ranking of refactoring patterns considering each quality attribute in isolation is the eigenvector of the respective pairwise matrix.

5. *Compute the overall ranking:* A ranking of the selected refactoring patterns is computed using the quality attributes eigenvector and the alternatives (refactoring patterns) eigenvectors. To compute the ranking of the refactoring patterns given the set of quality attributes, a matrix is created with the criteria (the quality attributes) and the alternatives (the refactoring patterns). This matrix is multiplied by the eigenvector of the quality attributes pairwise matrix.

This overall ranking can be used to focus the search for refactoring opportunities for the best ranked refactoring patterns, instead of looking for refactoring opportunities for refactoring patterns that contribute little to the overall software quality (i.e. are low ranked). Once a knowledge base containing the relationship between the refactoring patterns and the quality attributes is created, the developer has only to provide pairwise comparisons for the quality attributes, as the matrix manipulation activities can be automated.

Figure 4.1 shows the main concepts related to the creation of rankings of refactoring patterns. The *AHPRanking* class represents any AHP-based ranking. This class has two main methods: one for computing the ranking and another one for adding pairwise comparisons (represented by the *PairwiseComparison* class). Each pairwise comparison has two criteria (instances of classes realising the *Criterion* interface).

For ranking refactoring patterns, there is the need for creating three kinds of rankings. Rankings of quality attributes (represented by the *QAsRanking* class), rankings of refactoring patterns versus quality attributes (represented by the *RPvsQARanking* class), and an overall ranking (represented by the *RPsRanking* class). Each refactoring pattern in the overall ranking has a score and a position in the ranking (both computed by AHP).

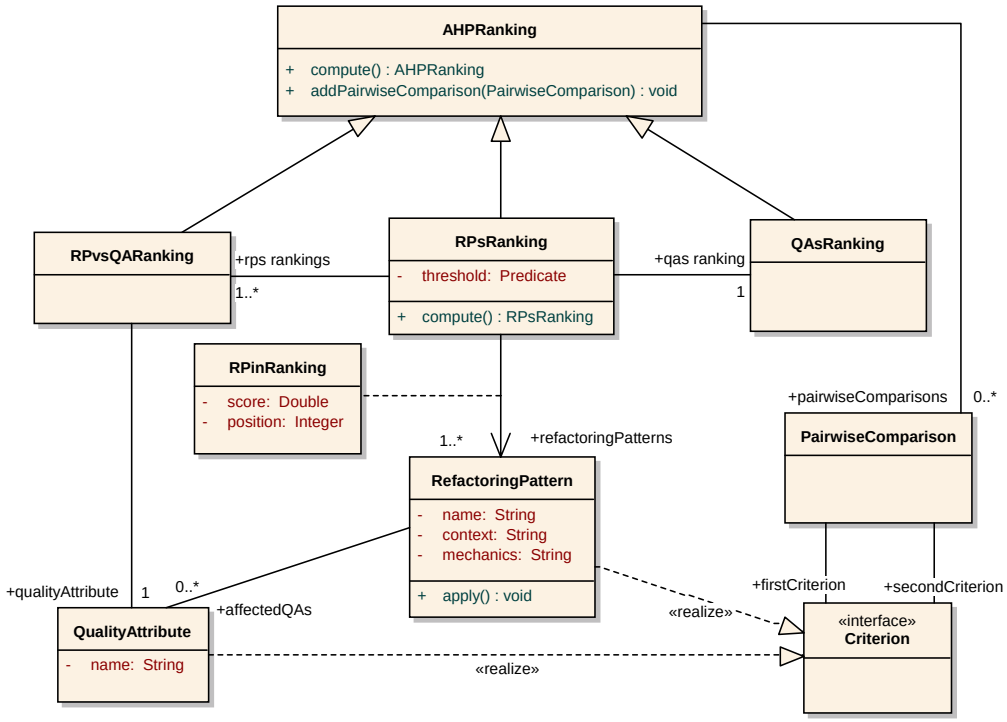

Figure 4.1: A conceptual model for ranking refactoring patterns

4.3 Case Study: Ranking Object-Oriented Refactoring Patterns with AHP

A ranking method is comprised by a set of steps to rank refactoring patterns according to a quality model. Below, each step of the AHP-based ranking method is shown using three quality attributes and four refactoring patterns. Section 4.3.1 describes the creation of the quality attributes pairwise comparisons. Section 4.3.2 exemplifies the creation of

the refactoring patterns pairwise comparisons. Section 4.3.3 shows the computed quality attributes ranking. Finally, Section 4.3.4 and Section 4.3.5 shows the refactoring patterns ranking and the overall ranking computation.

4.3.1 Creating the Quality Attributes Pairwise Comparisons

The importance of the selected quality attributes depends on the values of the development team, the process, the project and the organisation. Each project can have different sets and ordering of quality attributes.

In this example, the following quality attributes are considered: reusability, simplicity and comprehensibility. The following possible judgments for the selected quality attributes are expressed as pairwise comparisons:

- Simplicity is *moderately more important* than reusability and *slightly more important* than comprehensibility;
- Comprehensibility is *slightly more important* than reusability.

Note that these pairwise comparisons are hypothetical. Each project can have different needs in terms of quality attributes and can have different weights for each quality attribute. The responsibility for creating these comparisons can be delegated to a quality analyst, made by the project leader or by other means.

4.3.2 Creating the Refactoring Patterns Pairwise Comparisons

Several refactoring patterns are available in IDEs and modelling tools, including ones to move, rename, pull up, push down, extract or inline members of a class or a method, for example. Usually, the developers choose the refactoring patterns because of their availability within these IDEs and modelling tools.

For the example being developed, four different refactoring patterns were chosen, because they manipulate classes, methods and interfaces: *Pull Up Method*, *Rename Class*, *Inline Method*, and *Extract Interface*. They are compared and ranked in terms of the selected quality attributes.

The first step is evaluating each refactoring pattern in terms of each quality attribute, considering how much one refactoring pattern improves each quality attribute compared to the others. Let us suppose that the developer created a set of pairwise comparisons of the refactoring patterns in terms of the first quality attribute (simplicity), as follows:

- Pull Up Method is *strongly more important* than Rename Class, *weakly more important* than Inline Method and *strongly more important* than Extract Interface;
- Inline Method is *slightly more important* than Rename Class and *strongly more important* than Extract Interface;
- Rename Class is *weakly more important* than Extract Interface.

The second quality attribute is reusability, for which the following judgments are made to exemplify the process:

- Pull Up Method is *absolutely more important* than Rename Class, *moderately to strongly more important* than Inline Method and *slightly more important* than Extract Interface;

- Inline Method has the *same importance* than Rename Class;
- Extract Interface is *strongly more important* than Rename Class and Inline Method.

The third quality attribute is comprehensibility. For this quality attribute the following judgments are made using pairwise comparisons:

- Extract Interface is *slightly more important* than Pull Up Method and Rename Class and it is *absolutely more important* than Inline Method;
- Rename Class is *slightly more important* than Pull Up Method and *strongly more important* than Inline Method;
- Pull Up Method is *moderately more important* than Inline Method.

These pairwise comparisons can be inserted in a knowledge base to enable future reuse of the relations between refactoring patterns and quality attributes.

4.3.3 Computing the Quality Attributes Ranking

Using the pairwise comparisons defined in the previous section and the numerical values on Table 2.3, the quality attributes pairwise matrix $\mathcal{Q}$ is straightforwardly created from the defined pairwise comparisons:

$$\mathcal{Q} = \overbrace{\begin{bmatrix} 1.00 & 5.00 & 2.00 \\ 0.20 & 1.00 & 0.50 \\ 0.50 & 2.00 & 1.00 \end{bmatrix}}^{simp.\quad reus.\quad comp.} \begin{matrix} simp. \\ reus. \\ comp. \end{matrix}$$

This pairwise matrix is used to compute the weight of each quality attribute. In this matrix, the computed eigenvector $\mathcal{E}_q$ is:

$$\mathcal{E}_q = \begin{bmatrix} 0.5954 \\ 0.1283 \\ 0.2764 \end{bmatrix} \begin{matrix} simp. \\ reus. \\ comp. \end{matrix}$$

The $\mathcal{E}_q$ vector shows that the most important quality attribute in this setting is simplicity, then comprehensibility and last reusability. This setting can vary from project to project. The computed weights for each quality attribute define which are the preferred refactoring patterns for the selected quality attributes. In this case, the consistency ratio of the $\mathcal{Q}$ matrix is 0.48% (the matrix is consistent).

4.3.4 Computing the Refactoring Patterns versus Quality Attributes Ranking

The ranking of refactoring patterns for each quality attribute and the ranking of quality attributes can be automatically created as follows.

4.3.4.1 The Simplicity Ranking

First, the pairwise comparisons for the *simplicity* quality attribute are translated to their numerical equivalents and a pairwise matrix $\mathcal{S}$ is computed as follows:

$$\mathcal{S} = \overbrace{\begin{bmatrix} 1.00 & 3.00 & 7.00 & 7.00 \\ 0.33 & 1.00 & 2.00 & 7.00 \\ 0.14 & 0.50 & 1.00 & 3.00 \\ 0.14 & 0.14 & 0.33 & 1.00 \end{bmatrix}}^{pm \quad im \quad rc \quad ei} \begin{matrix} pm \\ im \\ rc \\ ei \end{matrix}$$

Here is the computed eigenvector of $\mathcal{S}$, named $\mathcal{E}_s$:

$$\mathcal{E}_s = \begin{bmatrix} 0,593 \\ 0,245 \\ 0,113 \\ 0,050 \end{bmatrix} \begin{matrix} pm \\ im \\ rc \\ ei \end{matrix}$$

In this case, *Pull Up Method* is the preferred refactoring pattern to be used, in terms of simplicity, when compared with the other three refactoring patterns (in order): *Inline Method*, *Rename Class* and *Extract Interface*. The consistency ratio is 5.83% and the pairwise matrix is consistent. Note that the difference between them is high. The developer can focus on those patterns with high values of relative importance for the quality attribute.

4.3.4.2 The Reusability Ranking

The reusability pairwise matrix $\mathcal{R}$, after translating the pairwise comparisons to their numerical equivalents, is:

$$\mathcal{R} = \overbrace{\begin{bmatrix} 1.00 & 6.00 & 9.00 & 2.00 \\ 0.17 & 1.00 & 1.00 & 0.14 \\ 0.11 & 1.00 & 1.00 & 0.14 \\ 0.50 & 7.00 & 7.00 & 1.00 \end{bmatrix}}^{pm \quad im \quad rc \quad ei} \begin{matrix} pm \\ im \\ rc \\ ei \end{matrix}$$

The computed eigenvector $\mathcal{E}_r$ is:

$$\mathcal{E}_r = \begin{bmatrix} 0.522 \\ 0.063 \\ 0.056 \\ 0.358 \end{bmatrix} \begin{matrix} pm \\ im \\ rc \\ ei \end{matrix}$$

Considering the ranking of refactoring patterns for reusability, the preferred refactoring pattern is *Pull Up Method*, followed by *Extract Interface*. The *Inline Method* and *Rename Class* refactoring patterns do not have much impact on this quality attribute. The consistency ratio is 2.52% and the pairwise matrix is considered consistent.

4.3.4.3 The Comprehensibility Ranking

The pairwise matrix $\mathcal{C}$ for comprehensibility is also created:

$$\mathcal{C} = \overbrace{\begin{bmatrix} 1.00 & 5.00 & 0.50 & 0.50 \\ 0.20 & 1.00 & 0.14 & 0.11 \\ 2.00 & 7.00 & 1.00 & 0.50 \\ 2.00 & 9.00 & 2.00 & 1.00 \end{bmatrix}}^{pm \quad im \quad rc \quad ei} \begin{matrix} pm \\ im \\ rc \\ ei \end{matrix}$$

And the computed eigenvector $\mathcal{E}_c$ is:

$$\mathcal{E}_c = \begin{bmatrix} 0.196 \\ 0.044 \\ 0.304 \\ 0.457 \end{bmatrix} \begin{matrix} pm \\ im \\ rc \\ ei \end{matrix}$$

Considering only comprehensibility, the best ranked refactoring pattern is *Extract Interface*, followed by *Rename Class* and *Pull Up Method*. The *Inline Method* refactoring pattern does not have much impact in terms of comprehensibility compared to the other ones selected. The consistency ratio is 1.9% (the pairwise matrix is consistent). Note that the order of the refactoring patterns is different, depending on the quality attribute.

4.3.5 Computing the Overall Ranking

For the example used in this chapter, the ranking is computed by:

$$\mathcal{O} = \overbrace{\begin{bmatrix} 0.593 & 0.522 & 0.196 \\ 0.245 & 0.063 & 0.044 \\ 0.113 & 0.056 & 0.304 \\ 0.050 & 0.358 & 0.457 \end{bmatrix}}^{simp. \quad reus. \quad compre.} * \overbrace{\begin{bmatrix} 0.5954 \\ 0.1283 \\ 0.2764 \end{bmatrix}}^{criteria}$$

Considering the initial quality attributes, the pairwise comparisons between them and the judgments for the alternatives, the ranking $\mathcal{O}$ of refactoring patterns is:

$$\mathcal{O} = \begin{bmatrix} 0.4743 \\ 0.1658 \\ 0.1583 \\ 0.2018 \end{bmatrix} \begin{matrix} pm \\ im \\ rc \\ ei \end{matrix}$$

The overall ranking shows that, considering the selected quality attributes, the selected refactoring patterns and the pairwise comparisons made, the best ranked refactoring pattern is *Pull Up Method*, followed by *Extract Interface*, *Inline Method* and *Rename Class*.

After the ranking is created, the developer can define a threshold to reduce the initial set of refactoring patterns to only those that have a ranking value higher than this threshold. He also can decrease the threshold value to add more refactoring patterns to the search for refactoring opportunities or increase it if the search for low ranked refactoring patterns is not being fruitful. The use of a threshold narrows the search for refactoring opportunities, focusing on the best ranked refactoring patterns.

The effectiveness of the refactoring patterns applied in each transformed software element can be evaluated using quantitative mechanisms, such as software metrics, impact functions (BOIS; MENS, 2003; BOIS, 2006) or qualitative evaluations.

4.4 Discussion

In this chapter, the evaluation of the refactoring patterns in terms of quality attributes is being defined using pairwise comparisons, provided by an individual developer or by a common agreement of a set of developers, for example.

This evaluation can be informal, when the pairwise comparisons are made using the comparative feelings expressed by the developers. One rather more structured alternative is to evaluate case by case each pair *refactoring pattern vs. quality attribute* using metrics to evaluate the impact of the refactoring patterns to the quality attributes.

For example, the developer can use a function to quantitatively measure how much the application of a given refactoring pattern changes a set of quality attributes. As it is not always possible to compute the function value for all refactoring patterns in advance (in some cases, the value depends on actual parameters), the developer can use the image of that function to obtain an *approximated* value. Also, qualitative evaluations can be made, using scenarios to assess the impact of each refactoring pattern in terms of the quality attributes.

Sometimes, however, the same refactoring pattern can lead to different effects on the quality attributes of software. For example, the *Extract Method* refactoring pattern can increase the comprehensibility of long methods, by extracting part of the behaviour to a new method, but it can decrease the comprehensibility of very small methods, by adding a new indirection to the real functionality.

In fact, each refactoring pattern can be applied to different contexts. If these different contexts are taken into account, the refactoring patterns can be associated with different predicates (for example, predicates to represent long methods or small methods). For instance, the developer can define the following predicate to express which methods are long:

$$extractLongMethod = (extractMethod(m), loc(m) > y) \tag{4.1}$$

In this case, m is a method, $locc(x)$ is a function that computes the number of lines of code of a method and y is a threshold. Each predicate can use one or more metrics to specify the cases in which it holds true.

Another issue is that when the developer creates a ranking to select a set of refactoring patterns, he does not know the *exact* impact of the application of those refactoring patterns. Such cases occur because the changes in the quality attributes and in their respective metric values are only known when the application of the refactoring pattern is effectively being conducted, with actual parameters in a software program. Chapter 7 describes how the effects of each refactoring pattern in quality attributes are computed for individual refactoring opportunities.

4.5 Tool Support

A proof-of-concept tool was developed to assess the practical use of the proposed approach. There is the need to provide pairwise matrices to express the relative importance of each quality attribute over the others and of each refactoring pattern over the others (for each quality attribute).

The tool converts these pairwise matrices to the equivalent AHP numerical representation, creates the pairwise matrices, computes the eigenvectors, elaborates the quality attributes ranking and the rankings of refactoring patterns regarding each quality attribute. These rankings are used to automatically compute the overall ranking. If the refactoring patterns matrices are created by a tool provider, for example, the user has only to provide the pairwise comparisons between the quality attributes.

The tool is comprised by a core module, which provides support for AHP, independently of for which purpose the ranking is being created. The developers can extend this

core module to provide support for specific rankings, such as the creation of rankings of refactoring patterns.

The core module is comprised by a *Ranking* class, which has the responsibility of grouping the ranking criteria, the pairwise comparisons and the pairwise matrices for the ranking, by a *Criterion* class, which can represent each criteria of the ranking, and by a *PairwiseComparison* class, which enables the creation of comparisons between two criteria using a numeric scale defined in the *Importance* class. Figure 4.2 shows a class diagram with the main classes of the module, their relationship and their public methods.

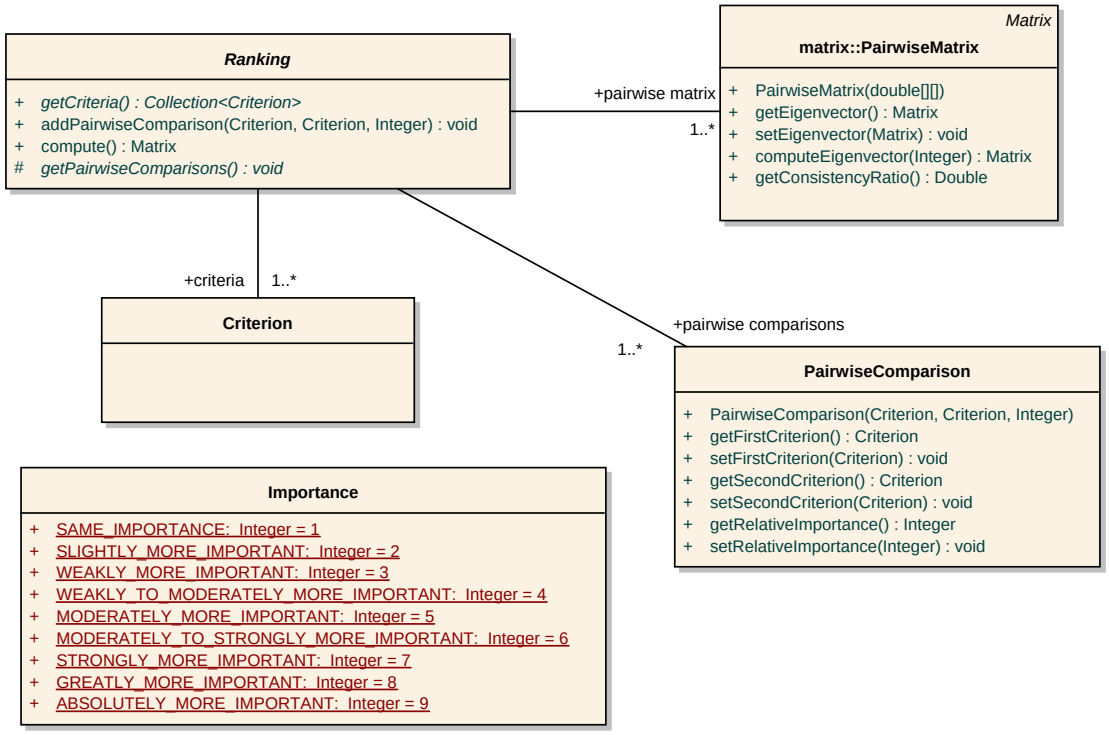

Figure 4.2: Tool for creating AHP rankings of refactoring patterns: core module

A set of additional classes were created for the example of this chapter: one representing a ranking of quality attributes (*QualityAttributeRanking*), another one for the ranking of refactoring patterns (*RPRanking*), and one class for each matrix representing the relationship of a quality attribute with the refactoring patterns. Figure 4.3 shows the relationship of these additional classes with the classes defined in the core module.

For example, the class representing the simplicity ranking contains only the pairwise comparisons between the simplicity quality attribute and the refactoring patterns:

```
1  public class SimplicityRanking
2    extends RPRanking{
3     protected void getPairwiseComparisons() {
4       addPairwiseComparison(PULL_UP_METHOD,
5         RENAME_CLASS,
6         Importance.STRONGLY_MORE_IMPORTANT);
7       addPairwiseComparison(PULL_UP_METHOD,
8         INLINE_METHOD,
9         Importance.WEAKLY_MORE_IMPORTANT);
10        ...
```

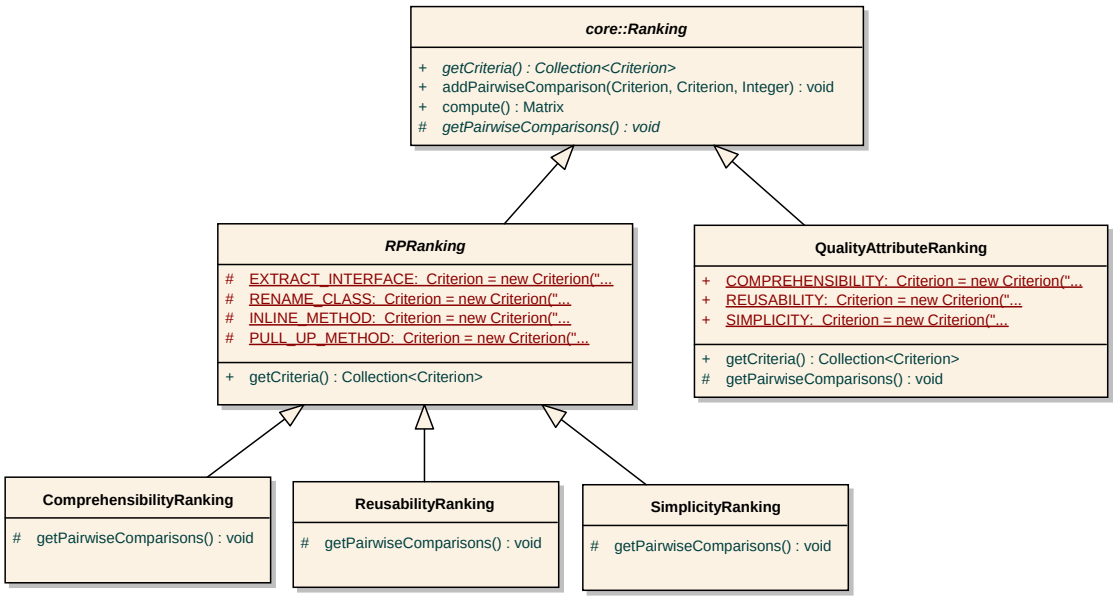

Figure 4.3: Tool for creating AHP rankings of refactoring patterns: example of use

```
11        }
12    }
```

Note that the ranking does not need to be updated frequently. Once the matrices relating refactoring patterns to quality attributes are created (by a tool provider, for example), the developer only has to inform the pairwise comparisons for the relative importance of each quality attribute over the others. The following change scenarios can affect the current ranking:

- *Changes in the pairwise comparisons:* The developer informs the changes in the pairwise comparisons and the tool computes the new ranking automatically.
- *The addition of a new Refactoring Pattern:* The developer should inform the pairwise comparisons of the new refactoring pattern with the old ones regarding each quality attribute. The tool then can automatically compute the new ranking.
- *The addition of a new Quality Attribute:* In this case, the developer should create a pairwise matrix comparing the refactoring patterns according to the new quality attribute. The tool then can automatically compute the new ranking.

4.6 Related Work

Du Bois and Mens (BOIS; MENS, 2003), (BOIS, 2006) suggest the application of refactoring patterns by specifying conditions in which their application can minimize coupling and maximize cohesion. Their formal analysis can be used together with the approach of this chapter to provide additional information for the developer to express the relative importance of refactoring patterns over the quality attributes using pairwise comparisons.

Tourwe and Mens (TOURWE; MENS, 2003) identify occurrences of shortcomings and propose the application of refactoring patterns using logic meta programming. Their

approach can be used together with the approach of this chapter to improve the results by focusing on refactoring patterns that improve the set of quality attributes selected by the developers.

Mens et al. (MENS et al., 2003) state that an open problem is to assess the effects of a refactoring pattern on software quality. By classifying refactoring patterns in terms of the quality attributes they affect, the effect of a refactoring on the software quality can be estimated. This chapter provides a quantitative approach to rank a set of refactoring patterns according to the quality attributes that the developers are concerned about.

4.7 Conclusions

Usually, there is room for improvements in existing software projects. However, resources are finite and must be directed to those activities that bring more benefits to the project. Considering refactoring activities, the developers should focus on the search for refactoring opportunities for those refactoring patterns that are more likely to improve the software being developed or maintained.

The use of AHP can help the developers to express the relationship between quality attributes and refactoring patterns and quality attributes between them. These relations are used to compute a ranking of refactoring patterns (according to the selected quality attributes), which can be used to focus the effort of searching for refactoring opportunities for those refactoring patterns that can have more impact in the developers quality attributes.

Without focusing in a restricted set of refactoring patterns, the developers can be losing time with refactoring opportunities that bring little to the overall quality of the software being developed. The techniques described in this chapter are adaptable: the quality attributes, the refactoring patterns and the weights can be changed and a new ranking computed automatically.

5 SHORTCOMINGS IN ASPECT-ORIENTED SOFTWARE

This chapter presents a catalogue of shortcomings that can occur in aspect-oriented software. It is organised as follows. Section 5.1 introduces the motivating scenario for the definition of a catalogue of shortcomings. Section 5.2 defines and details a number of shortcomings in aspect-oriented software, as well as which refactoring patterns can be used to reduce their impact. Section 5.3 describes related work and Section 5.4 summarises the chapter.

5.1 Introduction

Aspect-oriented software development aims at improving the separation of concerns by providing abstraction and composition mechanisms that deal specifically with the modularization of cross-cutting concerns (KICZALES et al., 1997). The most common abstraction mechanisms are aspects, pointcuts, advices and inter-type declarations. Although the use of aspects might help in the modularization of cross-cutting concerns, their use can introduce shortcomings either particular to the use of aspects, such as anonymous pointcut definitions or abstract method declarations, or similar to those found in objects, such as: pieces of code abandoned in a module and no longer used, code duplication and classes with too many or too few responsibilities.

A shortcoming[1] is a deficiency, inadequacy or incompleteness that a software element can have. The occurrence of a shortcoming can indicate a potential source of problems in software applications (ELSSAMADISY; SCHALLIOL, 2002). Such occurrences can be removed or alleviated by using appropriated refactoring patterns to change the software application. These shortcomings usually make it difficult to maintain and to reuse code in several development process activities (BOEHM; SULLIVAN, 2000) and can be alleviated by the identification of their symptoms and the removal of their causes.

There are catalogues and descriptions of shortcomings for object-oriented software (FOWLER et al., 1999; MONTEIRO; FERNANDES, 2005a), but their cataloging and detection in aspect-oriented software is still not explored enough.

Monteiro and Fernandes discuss shortcomings that arise in object-oriented software (MONTEIRO; FERNANDES, 2005a), indicating refactoring opportunities for code extraction from objects to aspects, without extensively discussing shortcomings that occur in aspect-oriented software. However, there are no available mechanisms to automatically detect their occurrences, neither detection tools for aspect-oriented software.

The main goal of this chapter is to describe a catalogue of shortcomings which can

[1] These shortcomings are also called *bad smells* (FOWLER et al., 1999). The former term was chosen in this thesis for the sake of aesthetics.

occur for aspect-oriented software. In this catalogue, shortcomings for object-oriented software are adapted to aspect-oriented software. This adaptation is done in such a way (i) to describe the problems that arise whenever those shortcomings are present in aspect-oriented software and (ii) to propose the application of refactoring patterns to help to remove those shortcomings. Each occurrence can be an opportunity to apply refactoring patterns.

5.2 Shortcomings in Aspect-Oriented Software

Shortcomings are a way to describe problems in existing software elements, by suggesting possible symptoms that can appear in the software elements, indicating areas that can be improved by the application of refactoring patterns. This application usually removes the causes of those shortcomings, in such a way that their effects are minimized or removed.

Table 5.1 shows some of the typical shortcomings already catalogued in the literature, both for object-oriented software and aspect-oriented software.

This section adapts a collection of object-oriented shortcomings to the context of aspect-oriented software. The adaptation aims at describing the problems each shortcoming brings when present in aspect-oriented software, and at proposing the use of refactoring patterns to help to reduce their impact. The examples of this section are retrieved from the samples provided with the IBM implementation of AspectJ, version 1.1 (HILSDALE; KICZALES, 2001).

Each shortcoming is presented by a definition of the problem related to the shortcoming, refactoring patterns that can be used to minimize or remove its effects, an example of its presence.

5.2.1 Anonymous Pointcut Definition

Because advices are not named, it is sometimes necessary to rely on the pointcut definition to have an idea of the affected points. Using the pointcut definition predicate directly on the advice can reduce the advice legibility and hide the predicate intention.

To clearly define the intent of a pointcut, a name is defined and used in any advice that affects the join points available in the pointcut. The *Extract Pointcut* refactoring pattern (IWAMOTO; ZHAO, 2003) can be used to extract definitions from pointcuts declared directly in the advice.

5.2.1.1 Example

Consider a *Debug* aspect, part of an example named *Space War* (a spaceship and asteroids game (HILSDALE; KICZALES, 2001)). This aspect is responsible for keeping and displaying debug information. In this aspect, there is an *anonymous pointcut definition* in the *after* advice (lines 3-5). The pointcut definition should be extracted from the advice to provide a better understanding of the affected points.

```
1 aspect Debug {
2   after(Ship ship, SpaceObject obj)
3     returning : call(void Ship.
4       handleCollision(SpaceObject))
5     && target(ship) && args(obj) { ... }
6 }
```

Table 5.1: Examples of shortcomings

Shortcoming	**Description**	**Source**
Aspect Laziness	Aspects that do not encapsulates all its expected responsibilities, but pass them to the classes, using inter-type declarations.	(MONTEIRO; FERNANDES, 2006)
Aspect Interaction	When there is interaction between aspects, and this interaction can present conflicts between aspects which are not orthogonal.	(DOUENCE; FRADET; SUDHOLT, 2002)
Divergent Changes	Divergent changes, as described for object-oriented software, occur in cases in which whenever the developer has to make a change, he must change several pieces of code. For example, every time a new column is inserted in the database, several classes must be changed.	(FOWLER et al., 1999)
Duplicated Code	The occurrence of the same code or structure in a software program. It leads to additional efforts in software maintenance and evolution activities.	(FOWLER et al., 1999; KERIEVSKY, 2005)
Double Personality	It is similar to the large class shortcoming, but is found in classes that play multiple roles. Ideally, each class should play a single role, with a set of related responsibilities.	(MONTEIRO; FERNANDES, 2006)
Feature Envy	In the context of object-oriented programming, this shortcoming occurs when a method refers too much on data or behaviour of another class or aspect, instead of referring to members of its containing class.	(FOWLER et al., 1999)
Large Class	A bloated class, with too many lines of code. Usually such classes are hard to reuse.	(FOWLER et al., 1999; KERIEVSKY, 2005)
Long Method	A method with too many lines of code, reducing its legibility.	(FOWLER et al., 1999; KERIEVSKY, 2005)
Middle Man	When there is too much delegation from a class to other classes.	(FOWLER et al., 1999)
Obsolete Parameters	When a parameter of a method is not being used in the body of the method.	(TOURWE; MENS, 2003)
Speculative Generality	When the developers start to provide support for possible future requests, which will possibly never occur, such as creating abstract classes that are not doing much, unnecessary delegation, methods with unused parameters, or odd abstract names.	(FOWLER et al., 1999)
Unnecessary Code	When there are private methods and fields that are not being used in a class, unused local variables, unused parameters, and unused imports.	(FOWLER et al., 1999)

After the extraction, the affected points are clearly defined in a named pointcut (line 2). The *collision* pointcut provides a definition to the predicate that, in the previous example, was attached directly to the advice. It also improves communication and reusability, as

the pointcut can be reused (if desirable).

```
1 aspect Debug {
2   pointcut collision(Ship ship,
3     SpaceObject obj): call(void
4      Ship.handleCollision(SpaceObject))
5       && target(ship) && args(obj);
6   after(Ship ship, SpaceObject obj)
7     returning : collision (ship, obj){ ... }
8 }
```

5.2.1.2 Detection Rule

Anonymous pointcuts can be found by evaluating pointcut definitions that are associated with advices but are not named (i.e. the pointcut expression is directly defined in the advice). The following heuristic rule evaluates occurrences of the primitive pointcuts in AspectJ directly defined in advices. It can be defined as follows:

Definition 5.2.1 *Let* $A = \{$*call, execution, get, set, initialization, preinitialization, staticinitialization, handler, adviceexecution, within, withincode, cflow, cflowbelow, if*$\}$ *be the set representing all the primitive pointcuts in AspectJ that are not related to context exposure. Let* B *be the set of the tokens in a given pointcut expression associated with an advice. The pointcut definition is an anonymous pointcut definition if and only if the predicate* $\exists a \in A \; \exists b \in B | b = a$ *holds.*

5.2.2 Speculative Generality

Sometimes classes and aspects are created to handle future requirements. As the use of aspects makes simpler to postpone some design decisions, it is possible to remove features that are not used in the system.

To remove advice parameters that are not being used, apply *Remove Advice Parameter* [2]. The *Collapse Aspect Hierarchy* (GARCIA et al., 2004), *Delete Aspect*, and *Inline Aspect* refactoring patterns can be used to remove unused aspects. Aspects and pointcuts with strange names can be renamed to the current semantics using *Rename Aspect* (HANENBERG; OBERSCHULTE; UNLAND, 2003) or *Rename Pointcut* (GARCIA et al., 2004).

5.2.2.1 Example

Consider, for example, the *TemplateOperationMonitor* class, which was created to provide standard operation bindings for XML-defined aspects to override:

```
1 public abstract aspect TemplateOperationMonitor {
2     protected pointcut classControllerExecTarget();
3     protected pointcut classControllerExec(Object controller)
          :
4         classControllerExecTarget() && target(controller);
5
6     protected pointcut methodSignatureControllerExecTarget();
```

[2] This refactoring pattern has not been previously defined and consists of removing unused advice parameters from an advice declaration.

```
7       protected pointcut methodSignatureControllerExec(Object
            controller) :
8           methodSignatureControllerExecTarget() && target(
                controller);
9
10  }
```

This aspect is not used by other aspects and it is not extended by sub-aspects. It provides several pointcuts to support a possible future need. It is possible that this need can never be materialised. The aspect can be deleted using *Delete Aspect*.

5.2.2.2 Detection Rule

It is not always straightforward to find occurrences of the *Speculative Generality* shortcoming. One way to search for certain occurrences is to look for unused features both in aspects and classes.

The following heuristic rule, for example, search for occurrences of concrete aspects that do not affect any other aspects or classes.

Definition 5.2.2 *Let M be the set of modifiers of an aspect α. Let $cda(\alpha)$ be the number of modules affected by the aspect α. An aspect is detected as an occurrence of speculative generality if the predicate: abstract $\notin M \wedge cda(\alpha) = 0$ holds.*

5.2.3 Feature Envy

In AspectJ, pointcuts can be defined both in aspects and classes. If a class defined pointcut is used by just one aspect, it is interesting that the pointcut is moved from the class to the aspect that uses it. The same problem might occur in classes, whenever a class method refers more to fields and methods of another class than referring members of its containing class.

The *Move Pointcut* refactoring pattern can be used to move these pointcuts from classes to aspects. It can also be used to move pointcuts between aspects.

5.2.3.1 Example

Consider, for example, a *Ship* class, which implements a spaceship in the *SpaceWar* example (HILSDALE; KICZALES, 2001). This class contains a pointcut definition (lines 2-5) that is used only in the *EnsureShipIsAlive* aspect (lines 8-13).

```
1  class Ship extends SpaceObject {
2    pointcut helmCommandsCut(Ship ship):
3      target(ship) && (call(void rotate(int))
4       || call(void thrust(boolean)) ||
5       call(void fire()));
6  }
7  ------------------------------------------------
8  aspect EnsureShipIsAlive {
9    void around (Ship ship):
10     Ship.helmCommandsCut(ship) {
11     if (ship.isAlive()) { proceed(ship); }
12   }
13 }
```

By moving the pointcut definition to the aspect, the coupling between class and aspect is reduced, and the cohesion of the aspect is improved.

Alwis et al. (ALWIS et al., 2000) suggest that pointcuts can be used to group semantically related class members. It is recommended that these pointcuts are kept together with the classes containing these methods. However, this definition can hide the complete identification of all join points affected by a pointcut. The developer has to check the classes containing the pointcuts, every time he wants to understand the whole set of affected points.

5.2.3.2 Detection Rule

For example, consider the detection of the occurrences of feature envy related with the detection of pointcuts in classes can be conducted as follows. The following heuristic rule can be defined:

Definition 5.2.3 *A class suffers from the feature envy shortcoming if it contains pointcuts defined in its body.*

Note that this is a sub-case of feature envy. Other heuristic rules are needed to detect all the cases.

5.2.4 Abstract Method Introduction

Aspects can be used to add state and behaviour into existing classes. This is accomplished through inter-type declarations. These declarations allow methods and/or fields to be inserted in classes defined by the aspect. However, their can cause problems when abstract methods are inserted in application classes.

The use of such inter-type declarations forces the developer to provide concrete implementations to the introduced methods in every affected class and sub-classes. This dependency unnecessarily increases the coupling between the aspect and the affected classes.

The introduction of abstract methods through an inter-type declaration should be avoided, because it demands that every time a sub-class of the affected class is created, implementations should be provided for these methods. If it cannot be avoided, apply the *Change Method Signature* refactoring pattern to change the modifier of the method to remove the abstract keyword.

5.2.4.1 Example

The following example shows a *Billing* aspect (HILSDALE; KICZALES, 2001), which charges for telephone calls according to the type and length of a performed call. In line 2, an abstract method is introduced to the *Connection* class. This class is responsible for determining the charge that applies to the customer according to the call type. This method is called *callRate*.

Next, on lines 3 to 5, the implementation of *callRate* method should be provided to the direct sub-classes of *Connection*, called *LongDistance* and *Local*.

```
1 public aspect Billing {
2   public abstract long Connection.callRate();
3   public long LongDistance.callRate()
4     { return 10; }
5   public long Local.callRate() { return 3; }
6   after(Connection conn):
```

```
 7        Timing.endTiming(conn) {
 8        long time = Timing.aspectOf().
 9        getTimer(conn).getTime();
10        long rate = conn.callRate();
11        getPayer(conn).addCharge(rate * time);
12    }
13  }
```

Consider a case in which a developer adds a new sub-class of *Connection*, named *International*. Problems would emerge, since this new class does not implement the *callRate* method. So, the class developer should be aware of which aspects affect the code, and then, add methods to the aspect. This dependency increases the complexity of the solution. Considering that the aspects and classes can be implemented by different developers, the developer of the Connection sub-classes can be unaware of the existence of *Billing* aspects. The abstract inter-type method declaration can be removed.

5.2.4.2 Detection Rule

The following definition can detect inter-type declarations of abstract methods.

Definition 5.2.4 *An inter-type method declaration is abstract if its definition contains the abstract modifier.*

5.2.5 Lazy Aspect

This shortcoming, initially defined by Monteiro and Fernandes (MONTEIRO; FERNANDES, 2005a) and further developed in this section, happens if an aspect is too small that it is better to eliminate it (to reduce maintenance costs, for example). Sometimes, this size reduction is related to previous refactoring or to unexpected changes in requirements (changes planned that do not occurred, for instance). After refactoring, some classes or aspects can become smaller.

If an aspect does justify its existence, use *Collapse Aspect Hierarchy* (GARCIA et al., 2004). This refactoring pattern focuses in reducing the hierarchy tree, moving members from one aspect to its sub-classes or from sub-classes to the super-aspect. Other similar refactoring patterns that can be applied to move members to other aspects are: *Pull Up/Push Down Pointcut*, *Pull Up/Push Down Advice*, and refactoring patterns to move pointcuts and advices. Empty aspects can be removed with *Inline Aspect* [3].

Other case that can be an occurrence of the *Lazy Aspect* shortcoming is when aspects do not affect many modules. This case includes aspects that have too many fields and methods that are not dealing with crosscutting concerns, affecting few other modules. Part of the state and behaviour of this aspect can be moved to a class using the *Move Method*, and *Move Attribute* refactoring patterns, thus reducing the size of the aspects to the minimum needed to implement the crosscutting behaviour (as aspects are abstractions created to deal with crosscutting concerns). The aspect can use association mechanisms to access these features.

5.2.5.1 Example

Consider an aspect named *TraceMyClasses* (lines 1-4), responsible for implementing tracing in an application. This aspect defines which points in the application should be

[3] This refactoring pattern has not been previously defined, but can be considered equivalent to other *Inline* refactoring patterns, and consists in inserting the aspect code directly into classes.

affected by the tracing mechanism. Unless the *Trace* aspect is used by other sub-aspects or is part of a reusable aspect library, there is no need to extend the *Trace* aspect just to define the affected points (line 2-3). This can be accomplished directly in the super-aspect.

```
1 public aspect TraceMyClasses extends Trace{
2   pointcut myClass(): within(TwoDShape)
3   || within(Circle) || within(Square);
4 }
```

Moving the pointcut using *Pull Up Pointcut* to the sub-class enable the deletion of the *TraceMyClasses* aspect using *Delete Aspect*. The resulting Trace aspect now has the *myClass* pointcut as one of its members.

```
1 public aspect Trace{
2   ...
3   pointcut myClass(): within(TwoDShape)
4   || within(Circle) || within(Square);
5 }
```

5.2.5.2 Detection Rule

A sub-set of occurrences of lazy aspects can be detected using the number of crosscutting members of an aspect, as follows:

Definition 5.2.5 *The crosscutting members of an aspect are the collection of all advice, pointcuts, declare constructions and inter-type declarations directly defined in this aspect. An aspect is considered an occurrence of the Lazy Aspect shortcoming whenever it does not have any crosscutting members.*

Another subset can also consider the crosscutting degree of an aspect (cda) metric, which is the number of affected modules of an aspect. The heuristic rule can be defined as:

Definition 5.2.6 *An aspect is considered a Lazy Aspect whenever the predicate the cda value for this aspect is lower than a pre-defined threshold (a minimum recommended value for the cda metric - defined by the development team).*

5.2.6 Divergent Changes

Another shortcoming occurs when some pointcut definitions are almost identical, varying only in their modifiers or in small parts of their predicate. Every time the duplicated part of a pointcut is modified, the same must be done to all the others. *Extract Pointcut* (IWAMOTO; ZHAO, 2003) can minimize that problem.

5.2.6.1 Example

Consider the *Debug* aspect:

```
1 aspect Debug {
2    pointcut allConstructorsCut():
3       call((spacewar.* && !(Debug+ ||
4       InfoWin+)).new(..));
5    pointcut allInitializationsCut():
6       initialization((spacewar.* &&
```

```
7         !(Debug+ || InfoWin+)).new(..));
8     pointcut allMethodsCut():
9         execution(* (spacewar.* &&
10        !(Debug+ || InfoWin+)).*(..));
11 }
```

Part of the pointcuts defined in lines 2 to 5 are repeated. Every time one of the duplicated pointcut changes, all the other predicates should be changed accordingly. After applying the *Extract Pointcut* refactoring pattern to the duplicated expression, the *Debug* aspect is changed to the following:

```
1  public aspect Debug {
2     pointcut affectedJPs(): within(spacewar.* && !(Debug+ ||
          InfoWin+));
3     pointcut allConstructorsCut(): affectedJPs() && call(*.new
          (..));
4     pointcut allInitializationsCut(): affectedJPs() &&
          initialization(*.new(..));
5     pointcut allMethodsCut(): affectedJPs() && execution(*
          *.*(..));
6  }
```

The *affectedJPs* encapsulates the previously duplicated pointcut expression into a named pointcut, facilitating the maintenance of the *allConstructorsCut*, *allInitializationsCut* and *allMethodsCut* pointcuts.

5.2.6.2 Detection Rule

The case in which pointcut definitions are almost identical can be detected using a similarity function, which is a function that receives two pieces of information as a parameter, compare them, and expresses the similarity of those pieces using a numerical value. As there is no such function specially defined for pointcut definitions, the developer can use existing similarity functions available in the literature for comparing arbitrary strings, such as edit distance (HALL; DOWLING, 1980), N-gram (NAVARRO, 2001), Jaro (JARO, 1989), and TF-IDF (SALTON; MCGILL, 1986).

Definition 5.2.7 *Consider a similarity function $f(p1, p2)$, which compares two pointcut definitions and returns a numerical value expressing their similarity. Consider a threshold t, which defines the minimum value to consider that p1 and p2 are similar definitions. Two pointcut definitions suffer from the Divergent Changes shortcoming if the predicate $f(p1, p2) \geq t$ holds.*

5.2.7 Double Personality

Aspects dealing with more than one concern can be divided in as many aspects as there are concerns. This case of having multiple concerns encapsulated by the same aspect (or the same class) is called *tangling* (ELRAD; FILMAN; BADER, 2001). Tangling can decrease the legibility and reusability of an aspect as the developer, in order to modify one concern has to understand its relationship with other concerns in the same class (ELRAD; FILMAN; BADER, 2001). For example, it is easier to reuse a plain *Account* class, dealing only with typical operations such as withdrawal, transferences and deposits, than a class with those operations plus database access operations, logging, security operations, and exception handling.

This shortcoming is usually discovered when the developer finds several unrelated aspect members (fields, pointcuts, inter-type declarations) in the same aspect. If aspect members related to different concerns exist inside a class, use the *Extract Class* refactoring pattern (FOWLER et al., 1999). If these members are aspect exclusive structures, use the *Extract Aspect* refactoring pattern to deal with these related members (MONTEIRO; FERNANDES, 2005a).

When different concerns can be separated through inheritance, grouping related members, it is possible to use the *Extract Sub-Aspect* refactoring pattern.

5.2.7.1 Example

Consider an example in which the *Debug* aspect defines advices dealing with different concerns simultaneously. This aspect collects points regarding user interface modification (lines 2-3), changes in the registry contents (lines 4-7), and ship collisions (lines 8-11), among others. Although all of these features are related to system debugging, they can be divided in several aspects, each one with a different perspective on debugging. These aspects can inherit from the same super-aspect, which can be the *Debug* aspect itself.

```
1  aspect Debug {
2    after() returning (SWFrame frame):
3      call(SWFrame+.new(..)){...}
4    after(Registry registry) returning :
5      target(registry) && (call( void
6        register(..)) ||
7        call(void unregister(..))){...}
8    after(Ship ship, SpaceObject obj)
9      returning : call(void Ship.
10       handleCollision(SpaceObject))
11       && target(ship) && args(obj){...}
12 }
```

The following example shows a sub-aspect of *Debug* containing an advice responsible for manipulating the debugging of ship collisions (lines 2-5). Defining a *Collision* aspect enables the developer to separate the debugging responsibilities, focusing, in this case, only in the collision specific requirements. This separation also makes easier to reuse the *Debug* aspect, since it contains only basic debugging functionalities.

```
1  aspect Collision extends Debug{
2     after(Ship ship, SpaceObject obj) returning:
3        call(void Ship.handleCollision
4           (SpaceObject))
5        && target(ship) && args(obj){...}
6  }
```

This sub-aspect can be created by the application of the *Extract Sub-Aspect* refactoring pattern in the *Debug* aspect. The *Extract Sub-Aspect* refactoring pattern creates a new sub-aspect containing all the selected members of the ancestor aspect or class (aspects can extend classes). In this case, the new aspect is named *Collision* and there is only one moved member: the *after(Ship, SpaceObject) returning* advice. The same sub-aspect can also be created with a different sequence, which is comprised by the application of a *New Sub-Aspect* refactoring pattern followed by the application of a *Push Down Advice* refactoring pattern in the *after(Ship, SpaceObject) returning* advice.

5.2.7.2 *Detection Rule*

There is no automated way to measure how many concerns are being handled by a class or an aspect. So, the first step in the heuristic rule is to start searching for aspects with several cross-cutting members (advices, inter-type declarations, and declare constructions). Then, the aspects found are analysed to count how many concerns they encapsulate. Those that encapsulate more than one concern can be defined as occurrences of the *Double Personality* shortcoming.

Such rule combines the number of crosscutting members of an aspect with the number of concerns that the aspect is encapsulating, is expressed as follows.

Definition 5.2.8 *An aspect is considered an occurrence of the Double Personality if the number of concerns encapsulated by this aspect is higher than one.*

5.2.8 Code Duplication

One of the main motivations for aspect-oriented software development is the reduction of code duplication. By providing abstraction mechanisms for modularization of cross-cutting concerns, there is a tendency to reduce duplications, since concerns previously scattered throughout the abstractions of the application can now be encapsulated in a single aspect, or in a small collection of aspects. Even so, duplication may occur among advices, due to bad coding or documentation.

If code duplication occurs in different advices of the same aspect, the repeated code can be extracted using the *Extract Method* refactoring pattern (FOWLER et al., 1999). This refactoring pattern allows that all advices from which the code has been extracted to call a new method with the extracted code, instead. It is also possible, if several advices have code that is entirely identical, to combine their pointcuts, removing the redundant advice. For this, the *Combine Pointcut* refactoring pattern (IWAMOTO; ZHAO, 2003) can be used.

If code duplication appears in different aspects, the developer can choose one of the duplicated software elements to be referenced by the others, change the code manually to reference the chosen duplicate, and deleting the remaining duplicates with the *Delete Method*, *Delete Advice*, *Delete Inter-Type Declaration*, and *Delete Attribute* refactoring patterns, for example.

If the duplication appears in aspects that extend the same super-class or super-aspect by inheritance, the duplicated structure can be moved up in the hierarchy. Possible duplicated structures include fields, methods, advices, pointcuts, and inter-type declarations. To remove these duplications, the following refactoring patterns can be applied: *Pull Up Attribute* (FOWLER et al., 1999), *Pull Up Method* (FOWLER et al., 1999), *Pull Up Advice* (GARCIA et al., 2004), *Pull Up Pointcut* (GARCIA et al., 2004) and *Pull Up Inter-Type Declaration* (GARCIA et al., 2004).

5.2.8.1 *Example*

The following listing shows an aspect responsible for implementing a mechanism to trace object constructors and method calls of a specific class. This aspect, called *Trace*, has advices to show the state of the join points before and after they occur, which happens every time the predicate defined in the pointcut is satisfied.

```
1 abstract aspect Trace {
2   abstract pointcut myClass(Object obj);
```

```
3     pointcut myConstructor(Object obj):
4        myClass(obj) && execution(new(..));
5     pointcut myMethod(Object obj): myClass(obj)
6        && execution(* *(..))
7        && !execution(String toString());
8     before(Object obj): myConstructor(obj) {
9        traceEntry("" + thisJoinPointStaticPart.
10           getSignature(), obj);
11    }
12    after(Object obj): myConstructor(obj) {
13       traceExit("" + thisJoinPointStaticPart.
14           getSignature(), obj);
15    }
16    before(Object obj): myMethod(obj) {
17       traceEntry("" + thisJoinPointStaticPart.
18           getSignature(), obj);
19    }
20    after(Object obj): myMethod(obj) {
21       traceExit("" + thisJoinPointStaticPart.
22           getSignature(), obj);
23    }
24 }
```

It is important to note that the advice code corresponding to the pointcut *MyConstructor* (on lines 8 to 11 and 12 to 15) is identical to the one associated to *myMethod* (lines 16 to 19 and 20 to 23). This duplication can be removed by joining the predicates defined in lines 2-4 and 5-7. The new predicate associated to the pointcuts would be *myConstructor(obj)* || *myMethod(obj)*, which allows the removal of the duplicated advice, as shown below.

```
1  abstract aspect Trace {
2     abstract pointcut myClass(Object obj);
3     pointcut myConstructor(Object obj):
4        myClass(obj) && execution(new(..));
5     pointcut myMethod(Object obj): myClass(obj)
6        && execution(* *(..))
7        && !execution(String toString());
8     before(Object obj): myConstructor(obj) || myMethod(obj) {
9        traceEntry("" + thisJoinPointStaticPart.
10           getSignature(), obj);
11    }
12    after(Object obj): myConstructor(obj) || myMethod(obj) {
13       traceExit("" + thisJoinPointStaticPart.
14           getSignature(), obj);
15    }
16 }
```

5.2.9 Shortcomings and Refactoring Patterns

Table 5.2 summarises the previous discussion, associating the shortcomings that can be found in the presence of aspects, as well as a set of refactoring patterns that can be used to improve the software application being developed.

Table 5.2: Shortcomings vs. refactoring patterns

Shortcoming	**Refactoring Patterns**
Abstract Method Introduction	Change Method Signature
Anonymous Pointcut Definition	Extract Pointcut
Code Duplication	Extract Method Combine Pointcut Pull Up Attribute Pull Up Method Pull Up Advice Pull Up Pointcut Pull Up Inter-type Declaration
Divergent Changes	Extract Pointcut
Feature Envy	Move Pointcut
Large Aspect	Extract Class Extract Aspect Extract Sub-Aspect
Large Pointcut Definition	Extract Pointcut
Lazy Aspect	Collapse Aspect Hierarchy Inline Aspect
Speculative Generality	Remove Advice Parameter Collapse Aspect Hierarchy Inline Aspect Rename Aspect Rename Pointcut

5.3 Related Work

Simon et al. (SIMON; STEINBRUCKNER; LEWERENTZ, 2001) use metrics to detect shortcomings. In particular, the authors try to detect opportunities to apply the following refactoring patterns: *Move Method*, *Move Attribute*, *Extract Class* and *Inline Class*. The cohesion of methods and attributes inside the classes of a software application are mathematically evaluated. The results are converted to a three-dimensional Cartesian coordinate system, and then rendered visually.

Invariants are values that remain constant every time some piece of code is executed, and can be an indicative of possible application of refactoring patterns. The Daikon tool (KATAOKA et al., 2001) uses program invariant detection to find suitable applications of refactoring patterns. The detection process implicates in the instrumentation of the code for analysis during runtime, and the execution of a comprehensive set of tests, so the tool can analyse a wide range of interactions.

Tourwe and Mens (TOURWE; MENS, 2003) propose the use of logic meta-programming to detect shortcomings. Other researches (BALAZINSKA et al., 2000; DUCASSE; RIEGER; DEMEYER, 1999) are similar in that both attempt to find repeated sections of source code throughout a software application. The former approach focuses on Java code and thus involves the parsing of the code, while the latter tries to remain language independent, considering the source code only as text strings. A few other approaches to automate the detection of shortcomings in software applications are presented in a detailed survey (MENS; TOURWE, 2004).

5.4 Conclusions

The relationship between shortcomings and refactoring patterns is very important. The identification of these shortcomings provides evidences of problems in the application design or implementation, and the application of refactoring patterns can help to increase the quality of the code under investigation.

This chapter defines shortcomings that arises in aspect-oriented systems and suggests refactoring patterns that can be used to remove or minimize these shortcomings. It extends other works that aim at defining shortcomings in object-oriented code (FOWLER et al., 1999) and shortcomings in aspect-oriented code (MONTEIRO; FERNANDES, 2005a).

The main contribution of this chapter is the adaptation and discussion of several shortcomings in aspect-oriented systems, and the recommendation of tools for their removal or minimization. Examples of these shortcomings in the AspectJ language were presented and discussed.

Although the shortcomings discussed in this chapter are expressed as symptoms in a specific language, they can be easily adapted to other aspect-oriented languages. As the AspectJ model is basis for several aspect languages, it can be seen as a good starting point to the definition of shortcomings for aspect-oriented software systems.

6 SEARCHING FOR REFACTORING OPPORTUNITIES

This chapter is organised as follows. Section 6.1 describes the motivation for an approach to search for refactoring opportunities in software applications. Section 6.2 discusses the basic steps for searching for refactoring opportunities. Section 6.2.3 shows the application of the approach to an example. Section 6.3 shows a case study in which a sub-set of the shortcomings are detected using an Eclipse plug-in. Section 6.4 describes the implementation of such plug-in. Section 6.5 discusses a set of issues on using the proposed approach, including the search space reduction, successive application of refactoring patterns and implementation issues. Section 6.6 includes some concluding remarks.

6.1 Introduction

Several mechanisms to restructure object-oriented software by applying refactoring patterns have been proposed. These mechanisms deal with the reorganisation of class diagrams (SUNYE et al., 2001; MARKOVIC; BAAR, 2005; ZHANG; LIN; GRAY, 2005), use case diagrams (RUI; REN; BUTLER, 2003; YU; LI; BUTLER, 2004; XU et al., 2004), and other diagrams (BOGER; STURM, 2002), including refactoring patterns to move methods from one class to another, to pull up class members to a superclass, to transform relationships into inheritance, to merge classes or to split methods (FOWLER et al., 1999), for example.

In large scale software, several opportunities for applying refactoring patterns can be found. The difficulty is to determine which of these locations effectively improve the qualities the developers wish to satisfy. Without automated support, the following questions are hard to answer: *"Where to refactor?"*, *"Which refactoring patterns are applicable?"* and *"What are the benefits of applying the selected refactoring patterns?"*.

Furthermore, it is not practical to apply the patterns one by one to later evaluate the improvements. Existing work on the quantitative assessment of the benefits of refactoring focus on the use of *impact functions* (BOIS; MENS, 2003; BOIS, 2006) to evaluate, for a given refactoring pattern, the changes in the software metric values, without having to actually perform the modifications. However, an approach tailored to use such functions associated with a set of quality attributes to identify and prioritise refactoring opportunities in software applications does not yet exist.

This chapter describes an approach for the identification and prioritisation of refactoring opportunities in software applications, allowing developers to focus on the refactoring opportunities that maximize the quality attributes they are interested in. This chapter provides mechanisms to rank a set of refactoring opportunities according to the influence of the refactoring process on the metrics that are being used to assess the quality attributes in a project.

The approach is evaluated using an example for an object-oriented software application. Two quality attributes are assessed using a heuristic rule, and the search for refactoring opportunities is exemplified using two refactoring patterns in a software application composed of 200 classes.

Additionally, this chapter also describes a set of rules that can be used to identify occurrences of the catalogued shortcomings in aspect-oriented software. These rules are functions created to quantitatively evaluate a set of quality attributes of a given software element. These rules can use metrics to specify the cases in which a software element has a shortcoming. A case study is conducted to show how these rules can be used to search for refactoring opportunities of five shortcomings in three systems written in AspectJ. (KICZALES et al., 2001a).

6.2 Searching for Refactoring Opportunities

This section defines an approach to identify refactoring opportunities in software applications, divided in two stages: *heuristic rules definition* and *search for refactoring opportunities*. The first stage aims at providing *heuristic rules* to evaluate a model or the individual elements of a model. The second stage (*search*) uses the heuristic rules previously defined to identify and prioritise refactoring opportunities in a software application. The last stage is iterative and can be partially automated.

6.2.1 Definition of Heuristic Rules

Heuristic rules are methods to help to solve a problem, commonly informal (RUSSELL; NORVIG, 2002). They are usually an approximation of a knowledge being represented. In this thesis, heuristic rules quantitatively evaluate a set of quality attributes of a given element of a model, or to detect the occurrence of shortcomings in software elements.

A set of metrics can be used to quantitatively assess quality attributes. For certain quality attributes, there are already a set of recommended metrics. For example, Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) describe a set of metrics associated with the *complexity* quality attribute in object-oriented design, made up of the following metrics: weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, response for a class and lack of cohesion in methods.

If no suitable metrics are available for the quality attributes selected, the Goal/Question/Metric approach (GQM) (BASILI, 1992) can be used to list and evaluate potential metrics to measure the quality attribute. GQM is an approach to software metrics that defines a measurement model, comprised of three levels: conceptual level (goals), operational level (questions) and quantitative levels (metrics).

There can be several stakeholders in a software project. To define a heuristic rule, the developer has to select the most important quality attributes according to the stakeholders needs. These quality attributes will be quantitatively analysed in terms of the associated metrics. Figure 6.1 shows the basic activities that must be performed by the Quality Analyst to define one or more heuristic rules to evaluate the software application.

First, the Quality Analyst selects the quality attributes and metrics to use with the heuristic rules. One issue that the Quality Analyst must be aware of is the complexity of the metric collection process. He has to ensure that the selected metrics can be collected in the case tool using existing metric collectors or that the chosen metrics can be integrated with external metric collectors.

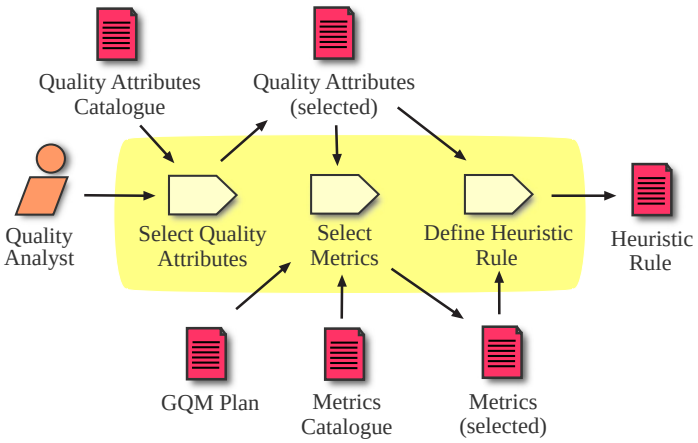

Figure 6.1: Heuristic rule definition activities

A heuristic rule provider can define a set of simple ready-to-use general heuristic rules, focusing on common quality attributes, such as complexity, reusability and modularity. The Quality Analyst can then extend the definition of these pre-defined heuristic rules or define new ones. If the Quality Analyst chooses to use existing functions, he has only to adjust the weights for the functions (defining the relative importance of each component of the heuristic rule over the others).

6.2.2 Activities

Using a set of heuristic rules the developer can:

1. Search for the software elements in the models described by the selected language or metamodel.
2. Check which refactoring patterns are applicable to the software elements found, and also if all the preconditions of the refactoring patterns are satisfied.

The following activities are needed to search for refactoring opportunities:

1. *Compute Metric Values:* The first step in finding refactoring opportunities is to compute the metric values for the software artefacts.
2. *Search for Applicable Refactoring Patterns:* In this activity, a subset of the applicable refactoring patterns is computed. This is done for the targets of each refactoring pattern and the pre-conditions defined for each transformation.
3. *Compute the value of the Heuristic Rules:* First, the heuristic rules values are calculated for the software application. These values will be compared to the heuristic rules values calculated using impact functions, which estimate the values of the heuristic rules after the application of refactoring patterns.
4. *Compare Heuristic Rules Values:* Using the values of heuristic rules before and after the application of refactoring patterns, the developer can compute the differences in the metric values.
5. *Prioritise and Suggest Refactoring:* This activity is the last step in the approach. The developer has a list of the differences in the heuristic rule values and a graphical comparison of the heuristic rule values to help choosing the most beneficial

refactoring patterns that can be applied to the software elements. The values of the metrics are the input to the heuristic rule. The refactoring patterns that provide results that are better than the current ones are presented to the developer. This refactoring process can be repeated until satisfactory results are achieved.

6.2.3 Example

This section illustrates the search for refactoring opportunities through an example. The source code of of the My SQL Connector J are used to search for refactoring opportunities. This software application is a native Java driver, which converts JDBC calls to native MySQL calls (MYSQL, 2009). The model of Version 5.0 contains 200 classes and the respective source code has 40.000 lines of code.

The search is conducted for opportunities to apply the *Extract Sub-Class* refactoring pattern and discuss opportunities found for the *Extract Class* refactoring pattern while doing the search for opportunities for an *Extract Sub-Class* application. A subset of the possible scenarios in which the *Extract Sub-Class* refactoring pattern can be applied is evaluated. In the example, the extraction of $2/3$, $1/2$, $1/3$ and $1/10$ of the methods to a new subclass are evaluated.

For this evaluation, the metric values for all the classes and subsequently the heuristic rule values are computed. For each class and each of the chosen scenarios, impact functions were applied to compute the predicted values of the heuristic rule in the My SQL Connector J models. Using the values one can see how the heuristic rule values change and which of the refactoring opportunities are more interesting that the others.

6.2.3.1 A Heuristic Rule for Simplicity and Reusability

This section shows an example of a heuristic rule definition, including the following activities: selecting quality attributes, selecting metrics, defining the heuristic rule and selecting the weights.

Selecting Quality Attributes.

First, the developer chooses the set of required quality attributes. The set of quality attributes depends on the software process, the project and the problem domain. For example, in embedded systems, memory is an important attribute, while in real-time systems, the timing constraint is the most important quality attribute. In other kinds of systems, such as information systems the main quality attributes can be reusability, usability and portability. The example uses *simplicity* and *reusability* to illustrate the approach, as both quality attributes are easy to understand and can be computed using traditional object-oriented complexity metrics.

Selecting Metrics.

Tsang et al. (TSANG; CLARKE; BANIASSAD, 2004) consider that *reusability* is the combination of weighted methods per class, depth of inheritance tree, number of children, coupling between objects, and lack of cohesion of methods. Fenton (FENTON; PFLEEGER, 1997) suggests the use of Chidamber and Kemerer metrics (CHIDAMBER; KEMERER, 1994) as indicators of software simplicity. This example uses three metrics to evaluate *reusability* and *simplicity*:

- **WMC.** The weighted methods per class (wmc) metric counts the number of operations in a given class (CHIDAMBER; KEMERER, 1994).

- **DIT.** The value for *depth of inheritance tree* (dit) is given by the longest path from a module to the class hierarchy root (CHIDAMBER; KEMERER, 1994).

- **NOC.** The *number of children* (noc) represents the number of direct sub-classes for a given class (CHIDAMBER; KEMERER, 1994).

Defining the Heuristic Rule.

The heuristic rule to be defined as example deals only with the metrics related to simplicity and reusability and can be defined as an aggregation function. To illustrate the process the weights 0.4 for simplicity and 0.6 to reusability were chosen for the heuristic rule, as follows:

$$f(x) = 0.4 * simplicity(x) + 0.6 * reusability(x) \tag{6.1}$$

The heuristic rule was normalised to return values in the [0,1] interval. The following set of assumptions was considered to define the heuristic rule used to illustrate the instantiation process:

- The smaller the values for wmc, dit and noc, the simpler is the design. This is in accordance with the view of Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) that states that the greater the noc value, the greater the likelihood of improper abstraction of the parent class.

- The higher the values for dit and noc greater the reuse, since inheritance is a form or reuse (CHIDAMBER; KEMERER, 1994). The more operations a class has, the less reusable it is (CHIDAMBER; KEMERER, 1994). Therefore, high values of wmc contribute to decrease reuse.

The simplicity and reusability functions are defined as a sum of products between the weights and the metrics. To normalize the values of each metric, the values are divided by the maximum value for this metric in the evaluated software application, denoted as $m(\mu)$, where μ is the metric and m is the function that computes a maximum determinated value for this metric:

$$\begin{aligned} simplicity(x) &= cw_1 * (1 - wmc(x)/m(wmc(x))) + \\ &\quad cw_2 * (1 - dit(x)/m(dit(x))) + \\ &\quad cw_3 * (1 - noc(x)/m(noc(x))) \\ reusability(x) &= rw_1 * (1 - wmc(x)/m(wmc(x))) + \\ &\quad rw_2 * (dit(x)/m(dit(x))) + \\ &\quad rw_3 * (noc(x)/m(noc(x))) \end{aligned}$$

Selecting the Weights.

This section uses the Analytical Hierarchy Process (AHP) (SAATY, 1990) to select the weights for the heuristic rule components. AHP focuses on finding an optimal solution using qualitative and quantitative decision analysis. The approach uses a set of pairwise comparisons to describe the relationship between two criteria and convert these comparisons to a weights vector.

For reusability, the following pairwise comparisons are used:

- dit is *slightly more important* than wmc;

- dit and noc have the *same importance*;

- noc is *slightly more important* than wmc.

Using the AHP approach to convert these pairwise comparisons to numerical values, the developer obtains the following weights vector: $V_r = \langle 0.2 \;\; 0.4 \;\; 0.4 \rangle$. The reusability function can then be expressed as:

$$\begin{aligned} reusability(x) &= 0.2 * (1 - wmc(x)/m(wmc(x))) + \\ & \quad 0.4 * (dit(x)/m(dit(x))) + \\ & \quad 0.4 * (noc(x)/m(noc(x))) \end{aligned}$$

The pairwise comparisons for simplicity are: (i) wmc is *weakly more important* than dit and than noc and (ii) dit is *slightly more important* than noc;

Converting these pairwise comparisons to a weights vector using AHP, gives the following vector for the simplicity quality attribute: $V_c' = \langle 0.5936 \;\; 0.2493 \;\; 0.1571 \rangle$. The simplicity function can then be defined as:

$$\begin{aligned} simplicity(x) &= 0.60 * (1 - wmc(x)/m(wmc(x))) + \\ & \quad 0.25 * (1 - dit(x)/m(dit(x))) + \\ & \quad 0.15 * (1 - noc(x)/m(noc(x))) \end{aligned}$$

6.2.3.2 Overview of the Refactoring Opportunities

The heuristic rule values before the refactoring and the heuristic rule values after the application of the *Extract Sub-Class* refactoring pattern are compared. Figure 6.2 shows the values of the heuristic rule for the following functions: the original heuristic rule, the value of the heuristic rule if $2/3$, $1/2$, $1/3$ and $1/10$ of the methods of each class are extracted to a new class. Note that the x-axis represents each class occurrence and the y-axis is the value of the heuristic rule for that particular class.

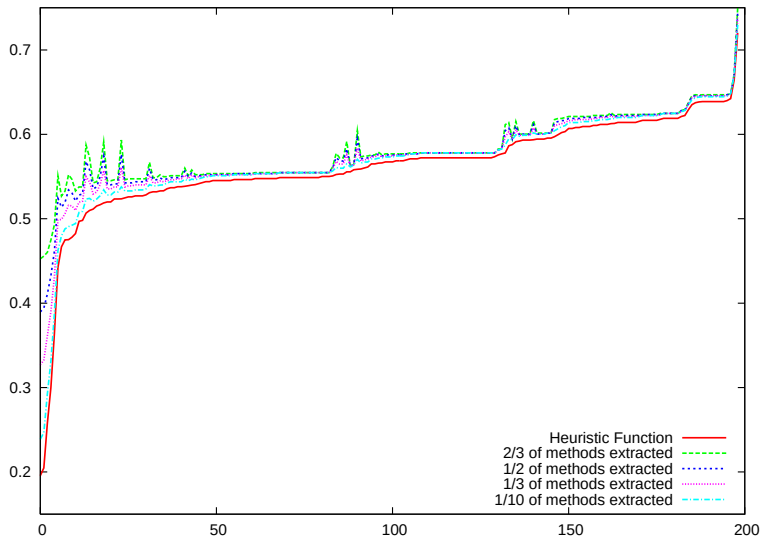

Figure 6.2: Examples of values for the heuristic rule if a subclass is extracted. The x-axis represents each class occurrence and the y-axis is the value of the heuristic rule.

Note that the differences of applying the *Extract Sub-Class* refactoring pattern are higher for those classes with a low heuristic rule value. These classes can be seen as the ones with the worst values for the heuristic rule. The more methods are extracted to the new subclass, the lower are the values for the heuristic rule of the original class (as shown in Figure 6.2).

In the next step, the refactoring opportunities are prioritised and the application of refactoring patterns is suggested. A threshold can be defined to choose how many classes will be inspected and analysed. One possible strategy is to inspect the classes in which the difference in the heuristic rule values before and after the application of refactoring patterns is higher.

6.2.3.3 Analysing Trade-offs and Prioritising Refactoring Opportunities

The ten top ranked classes are analysed in two ordered rankings: (a) a ranking containing the classes with the lowest heuristic rule values and (b) a ranking with the classes with the biggest differences in the heuristic rule values (before and after the application of the refactoring pattern).

Table 6.1 shows the ten classes with the worst values for the heuristic rule and Table 6.2 shows the ten classes with the larger difference between the values of the heuristic rule before and after a possible application of the *Extract Sub-Class* refactoring pattern. The developer can follow both lists in order to apply the refactoring patterns.

Table 6.1: Classes ordered by the values of the heuristic rule

Class	**wmc**	**dit**	**noc**	**H. Rule Value**
Connection	313	1	0	0.19
ConnectionProperties	305	0	5	0.20
ResultSet	246	0	1	0.24
DatabaseMetaData	211	0	1	0.28
UltraDevWorkAround	154	0	0	0.34
CallableStatement	128	2	0	0.43
MysqlIO	69	0	0	0.44
Statement	66	0	1	0.45
PreparedStatement	89	1	2	0.46
UpdatableResultSet	79	1	0	0.46

Figure 6.3 is a zoom in Figure 6.2, showing only the differences in the heuristic rule values for the first ten classes.

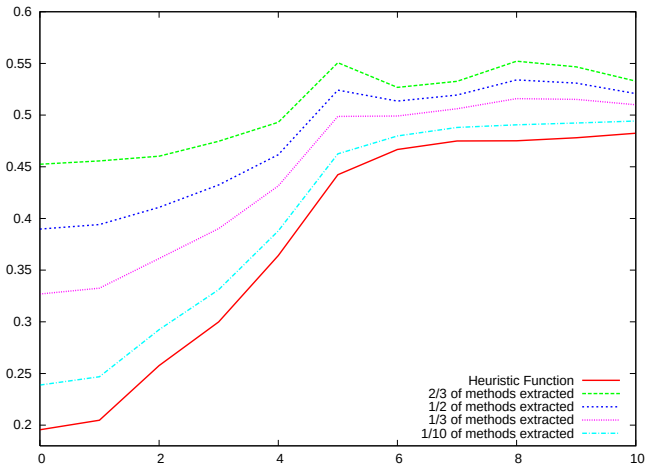

Figure 6.3: Examples of values for the heuristic rule if a sub class is extracted. The x-axis represents each class occurrence and the y-axis is the value of the heuristic rule.

Table 6.2: Classes ordered in a descendent way by the difference between the value of the heuristic rule before and after the application of refactoring patterns

		Δ changes in the heuristic rule values extracting x% of the methods			
Class	**H. Rule Value**	**x=2/3**	**x=1/2**	**x=1/3**	**x=1/10**
Connection	0.19	126%	95%	63%	21%
ConnectionProperties	0.20	120%	90%	60%	20%
ResultSet	0.24	79%	63%	42%	13%
DatabaseMetaData	0.28	61%	46%	32%	11%
UltraDevWorkAround	0.34	35%	26%	18%	6%
CallableStatement	0.43	23%	19%	12%	5%
StatementRegressionTest	0.50	14%	11%	7%	2%
PreparedStatement	0.46	13%	9%	7%	2%
ResultSetRegressionTest	0.51	15%	13%	9%	4%
CallableStatementWrapper	0.51	15%	11%	9%	2%

In the analysed tables, the first six classes are the same in both tables: *Connection*, *ConnectionProperties*, *ResultSet*, *DatabaseMetaData*, *UltraDevWorkAround* and *CallableStatement*. Other classes, such as *MysqlIO*, *Statement* and *UpdatableResultSet* appear as having low values for the heuristic rule. Although they do not appear in the top ten differences when applying the *Extract Sub-Class* refactoring pattern, these classes can also be restructured by the application of refactoring patterns, improving their heuristic rule values. The *StatementRegressionTest*, *ResultSetRegressionTest* and *CallableStatementWrapper* appear in the ten most benefited classes from the application of refactoring patterns in the list. Also, the *PreparedStatement* class appear in the two lists.

Note that the refactoring opportunities with the higher potential to improve the selected quality attributes are those that improve the heuristic rule values most. In Table 6.2, the opportunities that bring the higher improvements are those focusing the extraction of lots of methods and those applied to classes with a low value for the heuristic rule.

6.2.3.4 Inspecting the Top Ranked Refactoring Opportunities

Next, these classes are examined, looking for methods to be extracted to a new subclass. Large classes usually encapsulate more than one concern. Breaking the class into two or more classes can be advantageous in this case. Consider, for example the *Connection* class. It has the worst value for the heuristic rule and can have its metric values improved from the application of *Extract Sub-Class*.

Analysing this class, the developer can see that there is an inner class, named *UltraDevWorkAround* that has 154 methods. Extracting this class can improve the heuristic rule value. Note that when evaluating refactoring opportunities for the *Extract Sub-Class* pattern, sometimes the developer finds opportunities to apply *Extract Class* instead. In this case, it is useful to extract the inner class and *promote* it to a standalone class.

The second class detected as an opportunity for *Extract Sub-Class* is the *ConnectionProperties* class. When looking at the class diagram, the first thing to note is that this class has inner classes. Figure 6.4 shows the class with the attributes and methods compartments hidden to better visualize both the inner classes and two of its sub classes: *PropertiesDocGenerator* and *DocsConnectionPropsHelper*.

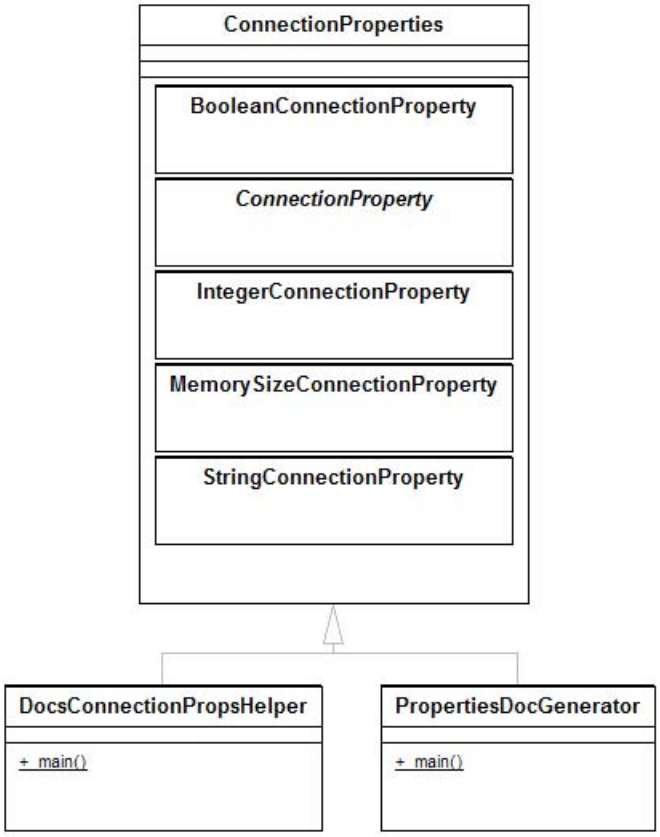

Figure 6.4: Class diagram for the *ConnectionProperties* class and some of its subclasses

In this case, several refactoring opportunities appear, not only to extract a class (or subclass). Also, the dependency between *ConnectionProperties* and the selected subclasses is very weak. The subclasses have only a *main* method and can be further inspected. As the source code of both classes is available, the *PropertiesDocGenerator* class is inspected:

```
1 public class PropertiesDocGenerator
2     extends ConnectionProperties {
3     public static void main(String[] args)
4         throws SQLException {
5         System.out.println(new
6             PropertiesDocGenerator().exposeAsXml());
7     }
8 }
```

This class does not need to extend the *ConnectionProperties* class. The inheritance dependency can be removed and an instance of the *ConnectionProperties* class can be created directly (instead of an instance of *PropertiesDocGenerator*).

The inspection of *DocsConnectionPropsHelper* class and *PropertiesDocGenerator* classes shows that they are equal, except for its name. None of them are being used by other classes and they can be both deleted.

Also, the inner classes are quite large and can be extracted to new classes. This can reduce the size of the *ConnectionProperties* class and reduce its complexity. Note that when one refactoring opportunity is found, other refactoring opportunities can be discovered by the developer by inspecting the models and the source code artefacts (if available).

The application of a refactoring pattern can create new classes (as occurs with the *Extract Sub-Class* pattern). The heuristic rule can be adapted to consider the number of classes as a metric to the expected quality attributes.

6.2.3.5 Applying Refactoring Patterns and Evaluating the Changes

A set of *Extract Class* and *Extract Sub-Class* refactoring patterns was applied in two of the classes in the initial list of refactoring opportunities. Table 6.3 shows the differences in the heuristic rule value for the two classes. In this case, the heuristic values in the *Connection* class improved in 94.74%, according to the new heuristic rule value. For the *ConnectionProperties* class the improvement is around 20%.

Table 6.3: The heuristic rule values after the application of *Extract Class/Sub-Class*

Class	before	after	Δ
Connection	0.19	0.37	+94.74%
ConnectionProperties	0.20	0.24	+20%

Note also that two classes were deleted (*DocsConnectionPropsHelper* and *PropertiesDocGenerator*) and five new classes were created. Table 6.4 shows the values for the metrics and the heuristic rule values for the new classes. These new classes do not have low values for the heuristic rules and are not in the top of the ranking when created. The focus must continue on those refactoring opportunities with low heuristic rule values. Opportunities for applying the refactoring patterns *Extract Class* and *Extract Subclass* in these new classes can be computed and they can enter in the ranking of refactoring opportunities regarding the difference between the values before and after a given refactoring pattern is applied.

Table 6.4: Metric and heuristic rule values for new classes

Class	wmc	dit	noc	heuristic rule
(+) ConnectionProperty	20	0	3	0.51
(+) BooleanConnectionProperty	7	1	0	0.54
(+) IntegerConnectionProperty	10	1	1	0.54
(+) MemorySizeConnectionProperty	3	2	0	0.57
(+) StringConnectionProperty	7	1	0	0.54

In those classes, the heuristic rule value of the original classes has improved. Two classes were deleted and five new classes were extracted from the *ConnectionProperties* class. The new classes have good values for the heuristic rule (comparing with the ones in the refactoring opportunities lists). The approach is iterative, whenever the developer applies refactoring patterns to some classes, the heuristic rules can be automatically recalculated and a new ranking of refactoring opportunities computed.

It is interesting to emphasize that, this example focuses on the *Extract Sub-Class* only, but other refactoring patterns were also applied to the classes (*Extract Class* and *Delete Class*). When more refactoring patterns are analysed, the number of refactoring opportunities increases and the rankings can help the developer to consider the most critical classes, instead of looking for all possible application of refactoring patterns. The use of the proposed mechanisms to automatically find the refactoring opportunities that are more suitable to the quality attributes he is interested is of great value. Additional heuristic rules can be used to evaluate the overall improvement of the application of *Extract Class*.

6.3 Case Study: Using Detection Rules to Search for Refactoring Opportunities in Aspect-Oriented Software

The following case study shows how a set of rules can be used to find shortcomings in well-known aspect oriented programs available as open source. The selected systems have different flavours of AspectJ programs as they include tutorial examples, academic software applications, open source software applications and commercial application of the language. The data was collected using an AST-based search engine for Java and AspectJ developed in the context of this thesis as a plug-in for the Eclipse development environment.

The first system selected is the collection of examples shipped with the AspectJ language reference implementation (IBM). These examples aim at showing the usage of the different constructions available to the language user. The second system is a collection of the GoF design patterns (GAMMA et al., 1995) implemented using AspectJ. This collection was developed by Hannemman and Kiczales (HANNEMANN; KICZALES, 2002) and it is used in other research papers on aspect-oriented metrics (GARCIA et al., 2005; MONTEIRO; FERNANDES, 2005a). The third system is a commercial product, developed by *GlassBox Corporation* and available as Open Source at Java.Net[1]. The *GlassBox Inspector* aims at delivering performance monitoring and troubleshoot mechanisms for *J2EE* applications using AspectJ and JMX[2].

Figure 6.5 shows some occurrences of shortcomings in the selected projects, produced with a detection plug-in program written to implement the rules defined in the previous section (Section 5.2). The view is populated whenever the user requires the activation of the detection plug-in for one or more AspectJ projects.

For each shortcoming, the following information is provided: project name, file name, type of shortcoming and additional details. The project name is the name given for the AspectJ project in the Eclipse IDE. File name describes the file name of the class or aspect. The type of shortcoming shows which shortcoming was detected. In this plug-in, a sub-set of the shortcomings described in the previous section are detected. The additional details column shows shortcoming-specific information for the user. For example, the occurrences of the *Anonymous Pointcut Definition* shortcoming show the pointcut expression in the details column, the name of aspects with the *Lazy Aspect* and *Double Personality* shortcomings are shown in the details column and occurrences of the *Abstract Method Introduction* shortcoming show the name of the abstract method being introduced.

In the following sections, each selected software application is detailed as following: first, a brief description about the software application under evaluation is presented; after that, a table summarizing the detected shortcomings is presented and each type of shortcoming is discussed regarding its occurrences in the application.

6.3.1 System 1: AspectJ Examples

The AspectJ examples provide illustrative source code to teach the users about the development of aspect-oriented programs using the language. These examples are divided into categories, such as: development aspects, tracing using aspects, production aspects and reusable aspects.

Each example works with different facets of the language. The domains used in those examples vary from telecom simulation and space war game to tracing systems. There

[1] `https://glassbox-inspector.dev.java.net/`
[2] `http://java.sun.com/products/JavaManagement/`

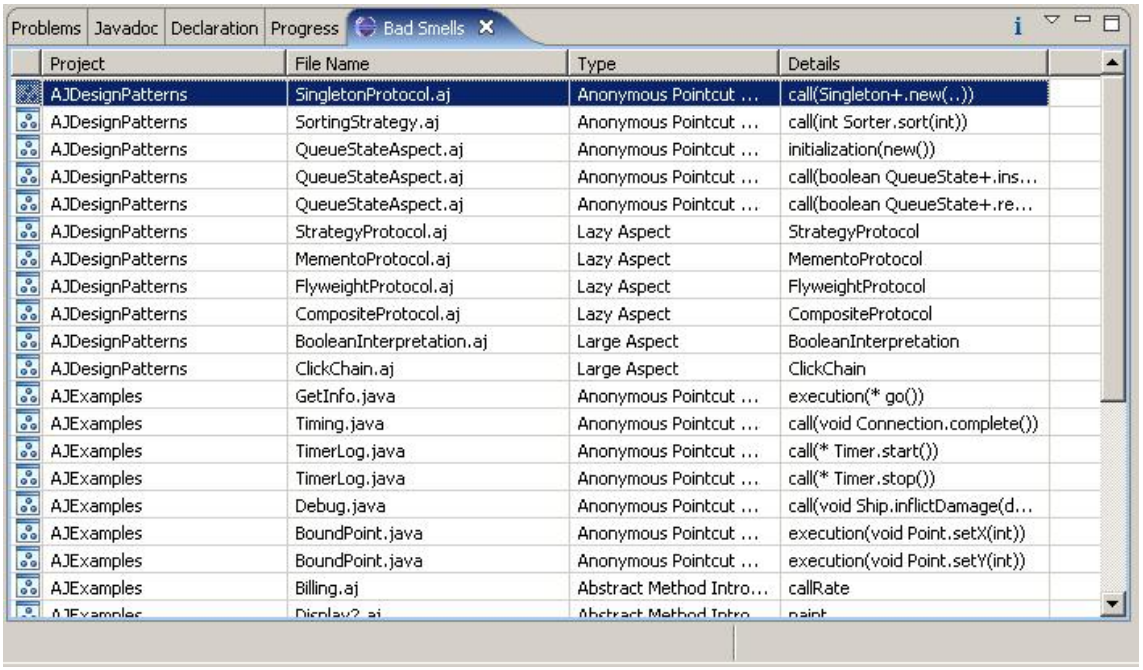

Figure 6.5: A view showing the shortcomings found in the projects

is also an implementation of a reusable Observer pattern (GAMMA et al., 1995) as an example.

In Table 6.5, the occurrences of each shortcoming are summarized. The *Anonymous Pointcut Definition* shortcoming is the one that appears most (22 cases). No instance of the *Lazy Aspect* shortcoming was found and an occurrence of the *Double Personality* shortcoming was detected in one of the examples. Occurrences of the *Feature Envy* and *Abstract Method Introduction* shortcomings appear in a few aspects.

Table 6.5: Shortcomings in AspectJ Examples

Type	**Number of Occurrences**
Classes	46
Aspects	27
Interfaces	5
Shortcoming	
Anonymous Pointcut Definition	22 of 52 advices
Double Personality	1 of 27 aspects
Lazy Aspect	0 of 27 aspects
Feature Envy	1 of 46 classes
Abstract Method Introduction	3 of 28 inter-type methods

As an example of an occurrence of the *Anonymous Pointcut Definition* shortcoming, the pointcut *demoExecs() && !execution(* go()) && goCut()* declared in an aspect named *GetInfo* is composed by two named pointcuts (*demoExecs* and *goCut*) and an anonymous pointcut definition (*!execution(* go())*). This last piece can be extracted into a new pointcut and its name used instead of the literal predicate. The resulting composition would be, for example: *demoExecs() && !goExecs() && goCut()*. Other detected occurrences of this shortcoming can be found in the *Timing*, *TimerLog*, *Debug* and *BoundPoint* aspects.

The high number of occurrences of this specific shortcoming is due to the nature of the

examples. Each example is intended to cover specific features of the language, without taking due reuse concerns in all applications. While good design techniques are desired, some of them can introduce unnecessary complexity to those that are trying to learn a new language (the main audience of the examples).

The aspect detected as an occurrence of the *Double Personality* shortcoming is the *Debug* aspect. It defines advices dealing with different concerns simultaneously. This aspect collects points regarding user interface modification, changes in the registry contents, and ship collisions, among other concerns. These features can be divided in several aspects, each one with a different perspective on debugging. Occurrences of the *Lazy Aspect* shortcoming were not found in the examples.

An occurrence of the *Feature Envy* shortcoming is present in the *Ship* class, which implements a spaceship in the *SpaceWar* example. This class contains a pointcut definition that is used only in the *EnsureShipIsAlive* aspect. The coupling between class and aspect is reduced, and the cohesion of the aspect is improved if the pointcut definition moves to the aspect.

An occurrence of the *Abstract Method Introduction* shortcoming exists in the *Billing* aspect, which charges for telephone calls according to the type and length of a performed call. So, the user of the class that receives the introduction should be aware of which aspects affect the code, and then, add methods to the aspect. This dependency can increase the complexity of the solution.

6.3.2 System 2: AspectJ Design Patterns

Hanneman and Kiczales (HANNEMANN; KICZALES, 2002) describe an experiment where the *gang of four* (GoF) design patterns (GAMMA et al., 1995) were implemented in both *Java* and AspectJ. The authors state that aspect-oriented implementations have improved modularity in 17 of the 23 studied cases.

The degree in which the enhancement occurs depends on the relationship among the roles played by the classes and objects within each pattern. Those patterns where an object plays more than one role, or where several objects play the same role, had the most significant improvement.

Garcia et al. (GARCIA et al., 2005) performed measurements on implementations of the GoF design patterns using quality metrics referring to separation of concerns, coupling, cohesion, and code size. The authors state that, in several cases, the aspect-oriented solution improved the separation of concerns relative to the participating roles of the design patterns.

Table 6.6 shows the occurrences of each type of shortcoming. The *Anonymous Pointcut Definition* shortcoming appears in five cases. Occurrences of the *Lazy Aspect* shortcoming were found four times and two occurrences of the *Double Personality* shortcoming were detected in the patterns. Occurrences of *Feature Envy* and *Abstract Method Introduction* do not appear in these examples.

A first occurrence of the *Anonymous Pointcut Definition* shortcoming occurs in the *SingletonProtocol* aspect: *call((Singleton+).new(..)) && !protectionExclusions()*. Instead, a composed pointcut can be used (*singletonCreation() && !protectionExclusions()*).

The second occurrence belongs to an aspect named *SortingStrategy*. The predicate contains a *call* primitive: *call(int[] Sorter.sort(int[]))*. This predicate affects only the calls to the *Sorter.sort* method. It appears in an around advice. The advice code can be inserted directly in the *sort* method. The same happens with the pointcut *initialization(new()) &&*

Table 6.6: Shortcomings in AspectJ Design Patterns

Type	**Number of Occurrences**
Classes	88
Aspects	42
Interfaces	16
Shortcoming	
Anonymous Pointcut Definition	5 of 15 advices
Double Personality	2 of 42 aspects
Lazy Aspect	4 of 42 aspects
Feature Envy	0 of 88 classes
Abstract Method Introduction	0 of 39 inter-type methods

target(queue) in the *QueueStateAspect*. The code triggered by the advice can be inlined in the constructor. Other examples of this shortcoming can be found in the *QueueStateAspect* aspect.

The *Lazy Aspect* shortcoming appears in four aspects: *StrategyProtocol*, *MementoProtocol*, *FlyweightProtocol* and *CompositeProtocol*. These aspects do not have any crosscutting members and can be safely converted to classes. Whenever an aspect does not have members implementing crosscutting concerns a class can (and should, if possible) be used instead.

The first occurrence of the *Double Personality* shortcoming is the *BooleanInterpretation* aspect. It is responsible for adding methods to perform the *replace* and *copy* operations in the following classes: *AndExpression*, *BooleanConstant*, *OrExpression*, *VariableExpression*, and *NotExpression*. To provide those methods, ten inter-type method declarations were used. The aspect can be broken in two aspects (one for the copy additions, another for the replace operations) or into five separated aspects: one for each affected class.

The second occurrence of the *Double Personality* shortcoming (named *ClickChain*) uses four parent declarations (*Frame*, *Panel* and *Button* implements *Handler* and *Click* implements *Request*) and defines inter-type declaration methods to add *handle* and *accept* behaviour to the *Button*, *Panel* and *Frame* classes. It also defines a pointcut to handle clicks in the *ChainOfResponsibility* pattern implementation. This aspect can be divided per affected classes (one aspect for affected class) or per operation (*handle* or *accept*). Occurrences of the *Feature Envy* and *Abstract Method Introduction* shortcomings were not detected in the examples.

6.3.3 System 3: Glassbox Inspector

The *Glassbox Inspector* project uses AspectJ and *JMX* to monitor performance for Java/J2EE applications. It provides information to identify specific problems, capture statistics, and monitor database calls. The version used in this case study was version 1.0 beta.

Table 6.7 summarizes the occurrences of shortcomings in the *Glassbox*. The *Anonymous Pointcut Definition* shortcoming occurs in seven aspects in the software application. Two occurrences of the *Double Personality* shortcoming and one occurrence of the *Lazy Aspect* shortcoming are present in the source code. The *Feature Envy* and *Abstract Method Introduction* shortcomings do not appear in these examples.

The first three *anonymous pointcuts* appear in the *TraceJdbc* aspect. The predicate

Table 6.7: Shortcomings in GlassBox

Type	**Number of Occurrences**
Classes	12
Aspects	26
Interfaces	7
Shortcoming	
Anonymous Pointcut Definition	7 of 27 advices
Double Personality	2 of 26 aspects
Lazy Aspect	1 of 26 aspects
Feature Envy	0 of 12 classes
Abstract Method Introduction	0 of 17 inter-type methods

call(java.sql..*(..)) || call(* javax.sql..*(..))* is the same in all advice. The predicate can be extracted in a single pointcut definition and the name of the new pointcut used in the advices. Other occurrences of the same shortcoming can be found in the *LogManagement*, *AbstractOperationMonitor* and *AbstractRequestMonitor* aspects.

Two aspects were detected as possible occurrences of the *Double Personality* shortcoming. The aspect named *LogManagement* has thirteen crosscutting members. Eight of them are inter-type method declarations that provide basic functionality for classes that should be logged. Instead of having methods such as: *logError(...)*, *logWarn(...)*, *logInfo(...)* and *logDebug(...)*, the developers can replace them by a general solution, passing the severity as a formal argument: *log(..., Severity severity)*.

The *ErrorHandling* aspect has eleven crosscutting members but does not need to be reduced. There is a pointcut named *handlingScope* that composes five other pointcuts and is used by an around advice. This advice ensures that errors in the monitoring code will not damage the underlying application code. As the pointcut predicate is a large one, the developers split the predicate into five others.

One occurrence of the *Lazy Aspect* shortcoming was detected in the Glassbox Inspector. The *AbstractResourceMonitor* aspect does not have crosscutting members, but it cannot be converted to a class because it extends the *AbstractRequestMonitor* aspect (in AspectJ, classes cannot extend aspects). The *Feature Envy* and *Abstract Method Introduction* shortcomings were not detected in the *Glassbox Inspector*.

6.4 Tool Support

A search engine was implemented to test the feasibility of the rules described in this chapter. It explores the AST support[3] available in the AJDT[4] project and is implemented as an *Eclipse* plug-in.

This plug-in extends both the Eclipse environment and the AspectJ environment. The AspectJ extension (Figure 6.6) was developed to provide mechanisms to find the shortcomings discussed in this chapter. The Eclipse extensions are available to provide visual information about the detected shortcomings.

Note that the developed tool focus on the detection of the shortcomings in AspectJ programs developed in the Eclipse environment. The tool can be integrated with the IDE

[3]The developments in the AST support are still in progress and they are covered by enhancement `https://bugs.eclipse.org/bugs/show_bug.cgi?id=110465`.

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

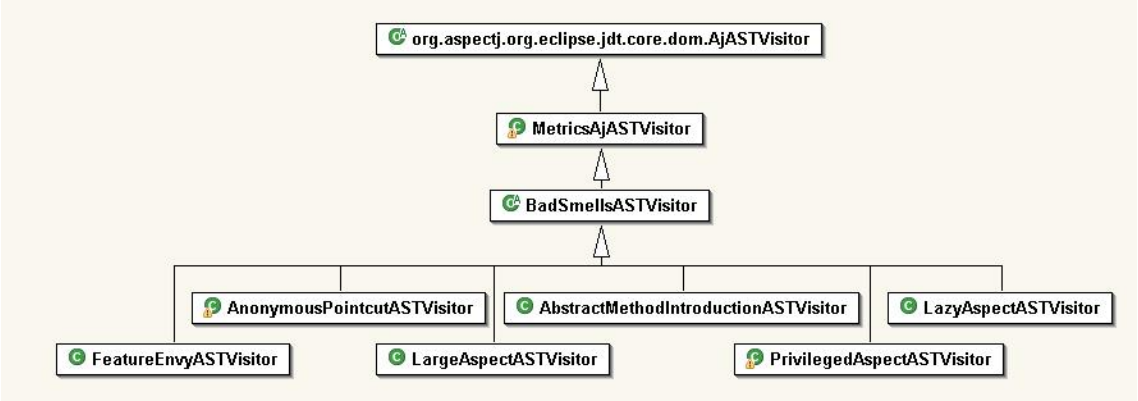

Figure 6.6: Class hierarchy of the AspectJ extension

support for applying refactoring patterns, enabling the developer to remove the detected shortcomings.

The classes of the AspectJ extension package are briefly described here:

AjASTVisitor This class implements a visitor for abstract syntax trees. For each different concrete AST node type T there are some methods that can be used, such as *visit(T node)* or *endVisit(T node)*, to visit a given node and perform some arbitrary operation. This class is provided by the AspectJ reference implementation.

MetricsAjASTVisitor This class collects meta-information about the visited AST. It holds data about advices, pointcuts, inter-type field declarations, inter-type method declarations, declare constructions and size related metrics.

BadSmellsASTVisitor This visitor is responsible for reading information from eclipse files and collecting data to be displayed in the user interface. It is the direct superclass of all the shortcomings AST visitors.

Other Classes There are other visitors defined to each type of shortcoming being detected. Examples are: *AnonymousPointcutASTVisitor*, *AbstractMethodIntroductionASTVisitor*, *LazyAspectASTVisitor*, *FeatureEnvyASTVisitor*, *LargeAspectASTVisitor* and *PrivilegedAspectASTVisitor*.

In the following sections, the implementation is briefly described.

6.4.1 Searching for Anonymous Pointcut Definitions

An implementation of a function to detect occurrences of the *Anonymous Pointcut Definition* shortcoming is shown below. First, a set named *primitive*, containing all pointcuts not concerned with context exposure is created (line *2*). After, the string *s* containing the pointcut predicate is divided into tokens (line *7*), which are individually compared with the *primitive* set. If the set contains *s*, the method returns *true*, *false* otherwise.

```
1 protected boolean isAnAnonymousPointcut(String s) {
2     Collection primitive = new ArrayList();
3     primitive.add("call");
4     primitive.add("execution");
5     ...
6     boolean temp = false;
```

```
7       String[] result = s.replace("("," ").replace(")"," ").split
            ("\\s");
8       for (int x=0; x<result.length; x++)
9           if (primitive.contains(result[x])){
10              temp = true; break;
11          }
12      return temp;
13      }
14  }
```

The detection of anonymous pointcuts in AspectJ and AJDT can be done using a visitor, which visits advice declarations looking for the use of anonymous pointcuts. Listing 6.1 shows the implementation of such visitor. It visits all *AdviceDeclaration, AfterAdviceDeclaration, AroundAdviceDeclaration* and *BeforeAdviceDeclaration* nodes. Whenever the function *isAnAnonymousPointcut* returns *true*, a *BadSmellsEvent* instance is created to gather information about the shortcoming (lines *9-11*).

Listing 6.1: An AST visitor that detects the anonymous pointcut shortcoming

```
1  public class AnonymousPointcutASTVisitor extends
       BadSmellsASTVisitor {
2      private boolean visitAdvice(AdviceDeclaration node) {
3          isAnAnonymous(node.getPointcut());
4          return false;
5      }
6      protected void isAnAnonymous(PointcutDesignator pd) {
7          if (pd instanceof DefaultPointcut)
8              if (isAnAnonymousPointcut(((DefaultPointcut)pd).
                   getDetail())){
9                  BadSmellsEvent event = new BadSmellsEvent();
10                 event.setType("Anonymous Pointcut Definition");
11                 ...
12             }
13         ...
14     }
15 }
```

6.4.2 Searching for Double Personality

The threshold can be defined by the user of the function, or given as a constant. The detection in AspectJ can be implemented as a visitor (see Listing 6.2). All *TypeDeclaration* nodes are inspected in the end of the visiting process (line *2*). Whenever the node is an aspect, the number of declared members is obtained and compared to the τ value, defined in a constant named *TAU* available in a class named *Consts* (lines *4-5*). If the number of crosscutting members is equal or higher than *TAU*, the aspect is marked as an occurrence of the *Double Personality* shortcoming, *false* otherwise.

Listing 6.2: AST visitor responsible for the detection of the *Double Personality* shortcoming

```
1  public class DoublePersonalityASTVisitor extends
       BadSmellsASTVisitor {
2      public void endVisit(TypeDeclaration node) {
```

```
3         super.endVisit(node);
4         if (((AjTypeDeclaration) node).isAspect())
5            if (getNumberOfMembers() >= Consts.TAU){
6               BadSmellsEvent event = new BadSmellsEvent();
7               event.setType("Double Personality");
8               ...
9            }
10    }
11 }
```

6.4.3 Searching for Lazy Aspects

To detect occurrence of the *Lazy Aspect* shortcoming, the *LazyAspectASTVisitor* creates shortcoming events whenever an aspect without crosscutting members is found (see Listing 6.3).

Listing 6.3: AST visitor responsible for the detection of the *Lazy Aspect* shortcoming

```
1  public class LazyAspectASTVisitor extends BadSmellsASTVisitor
       {
2     public void endVisit(TypeDeclaration node) {
3        super.endVisit(node);
4        if (((AjTypeDeclaration) node).isAspect())
5           if (getNumberOfMembers() == 0){
6              BadSmellsEvent event = new BadSmellsEvent();
7              event.setType("Lazy Aspect");
8              ...
9           }
10    }
11 }
```

6.4.4 Searching for Feature Envy

The implementation using AspectJ is pretty straightforward (see Listing 6.4). The program checks all nodes representing types (aspects, classes and interfaces) and verifies if a class does not implement a pointcut in its body. If this happens, an event is generated. Note that the *visit(PointcutDeclaration node)* method (line *5*) is executed only if the method *visit(TypeDeclaration node)* (line *2*) returns *true*.

Listing 6.4: AST visitor responsible for the detection of the *Feature Envy* shortcoming

```
1  public class FeatureEnvyASTVisitor extends BadSmellsASTVisitor
       {
2     public boolean visit(TypeDeclaration node) {
3        return (!((AjTypeDeclaration) node).isAspect() || node.
            isInterface());
4     }
5     public boolean visit(PointcutDeclaration node){
6        BadSmellsEvent event = new BadSmellsEvent();
7        event.setType("Feature Envy");
8        ...
9        return false;
10    }
```

```
11  }
```

6.4.5 Searching for Abstract Method Inter-Type Declarations

A class that detects this kind of shortcoming can be seen in Listing 6.5. A *visit* method is defined to visit all inter-type method declarations (line 2). If the *node* has *abstract* modifier, the inter-type declaration is abstract, *false* otherwise (line 4).

Listing 6.5: AST visitor responsible for the detection of the *Abstract Method Inter-Type Declaration* shortcoming

```
1  public class AbstractMethodIntroductionASTVisitor extends
       BadSmellsASTVisitor {
2      public boolean visit(InterTypeMethodDeclaration node){
3          String name = node.getName().toString();
4          if (Modifier.isAbstract(node.getModifiers())){
5              BadSmellsEvent event = new BadSmellsEvent();
6              event.setType("Abstract Method Introduction");
7              ...
8          }
9          ...
10     }
11 }
```

6.5 Discussion

6.5.1 Reducing the Search Space

The developer can use the computed ranking together with the number of total targets to decide which patterns will be used to search for refactoring opportunities. Also, the developer can define additional constraints to further reduce the search space. For example, the developer can search for refactoring opportunities of *Extract Interface* with zero, 50% of the methods and 100%, to reduce the total number of targets.

In this case, the proposed ranking can be, beyond looking for refactoring opportunities to those best ranked refactoring patterns to search also for refactoring patterns that enable the application of the best ranked refactoring patterns and to take into account refactoring patterns that disable the application of the best ranked refactoring patterns. This search can be extended to several levels of application of refactoring patterns, denoted by n. The developer could focus on:

1. Search for refactoring opportunities for all refactoring patterns in a catalogue
2. Search for refactoring opportunities for the $\mathcal{X}$ best ranked refactoring patterns
   (a) Also, search for refactoring opportunities that enable the application of the $\mathcal{X}$ best ranked refactoring patterns (considering n levels)
   (b) Also, take into account refactoring opportunities that disable the application of the $\mathcal{X}$ best ranked refactoring patterns (considering n levels)

Other considerations to reduce the search space can be searching for refactoring opportunities considering:

- A sub-set of packages;
- A sub-set of classes;
- A sub-set of refactoring patterns;
- A sub-set of some elements of the metamodel;
- Modification date (or version);
- Software elements already restructured by the application of refactoring patterns.

These criteria can also be used to group the refactoring opportunities, so the developers can focus on the system areas that have the worst values for the heuristic rules, the highest improvement in terms of the quality attributes or other criteria defined by the SQA team, for example.

6.5.2 Dealing with Successive Refactoring

Considering the use of this approach considering successive or composite refactoring patterns, the following comments are made. Composite refactoring patterns can be treated like all the other refactoring patterns. The successive application of refactoring patterns can be explored to search refactoring opportunities for more than one level of refactoring. For example, instead of searching only for the application of a *Pull Up Method* refactoring pattern, the developer can also search for opportunities that are created with the new method in the super class.

This case leads to another strategy for the ranking composition. The developer can also consider in the ranking of refactoring opportunities a set of *enabling refactoring patterns*, which are those refactoring patterns that do not improve directly the selected quality attributes but creates a refactoring opportunity for a well ranked refactoring pattern that did not exist before the application of the enabling refactoring pattern. Also, critical pair analysis (MENS; TAENTZER; RUNGE, 2005) can be used to spot occurrences of conflicts between refactoring patterns, i.e. refactoring patterns that *disable* other refactoring patterns.

6.6 Conclusions

There are several opportunities for refactoring in software applications, aiming at improving the quality attributes the developers cares about. Identifying those opportunities and evaluating the effects of software transformations manually can be ineffective and error-prone. An approach is defined to identify and prioritise refactoring opportunities in software applications and exemplify its use by an instantiation of this approach to identify refactoring opportunities in UML class diagrams. The results of the example shows that the use of a multi-criteria heuristic rule to find opportunities for refactoring can help the developer to focus on the refactoring patterns that bring the most benefits.

This process can be instantiated to other metamodels, quality attributes, refactoring patterns and metrics. It comprises two stages: preparation and search. Automated tools for both stages can enable the developers to configure the relative importance of the quality attributes and to focus on the task of analysing the identified refactoring opportunities and restructuring the software models accordingly.

This chapter also discussed some rules to detect shortcomings in AspectJ programs. The defined rules can be extended to deal with more cases of each shortcoming. The provided implementation can be extended to support those other shortcomings. Additional software systems can be the subject of further investigation. The appropriate detection and removal of shortcomings can affect quality attributes in the software application being modified and each refactoring pattern application might be evaluated regarding those attributes.

The evaluated systems in the case study have, in general, a low number of shortcomings. The one that appears more frequently is the *Anonymous Pointcut Definition* shortcoming. This shortcoming is usually removed whenever the predicate is used in more than one advice/inter-type declaration or when the aspect is an abstract one. Occurrences of shortcomings such as *Feature Envy* and *Abstract Method Introduction* were less frequently detected in the examined applications. The detection of the *Double Personality* shortcoming depends on the definition of significant thresholds, which can be gathered from the analysis of existing systems or provided by the users of the detection tool and the carefully analysis of the number of encapsulated concerns. Most of the occurrences of the *Lazy Aspect* shortcoming in the evaluated systems are associated with aspects that do not have crosscutting members and can be replaced by classes.

The use of such approach in early stages of a software development process can be beneficial. Both design and analysis models can be evaluated using the proposed rationale. As the number of models and model elements can be large, it is interesting to focus on the more problematic elements, according to the quality attributes that the development team is concerned about. Future work focuses on integrating and automating the process in current software modelling tools.

7 EVALUATING THE EFFECTS OF REFACTORING

This chapter describes how to create impact functions to quantitatively evaluate the effects of refactoring on software quality and how to use such functions. It is organised as follows. Section 7.1 describes the need for mechanisms to evaluate the effects of refactoring on software quality. Section 7.2 describes an approach for the creation of impact functions. Section 7.3 exemplifies these impact functions for an aspect-oriented refactoring pattern. Section 7.4 shows a case study that presents how impact functions can be used in a software application. Section 7.5 describes tools support. Sections 7.6 and 7.7 present related work and conclusions, respectively.

7.1 Introduction

In the context of aspect-oriented software (KICZALES et al., 1997), a number of refactoring patterns can be used to manipulate both classes and aspects (MONTEIRO; FERNANDES, 2005a, 2006). Specifically, refactoring patterns that deal with aspect-oriented software allow moving members: (a) from classes to aspects, (b) between aspects and (c) from aspects to classes. These refactoring patterns provide mechanisms to reorganise the overall structure of models and source code.

However, it is not always clear to the developer how the software elements being restructured are affected by refactoring. When a software application is restructured, several metric values change and it is interesting to know in advance the effects of each transformation (KATAOKA et al., 2002). Current research on the evaluation of the effects of refactoring in software elements focuses on the creation of *impact functions* (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006) to predict changes in metric values when a refactoring pattern is applied to those artefacts. Unfortunately, there is no explicit approach for the creation of these impact functions, and there are no defined impact functions for aspect-oriented software.

This chapter describes an explicit approach to create impact functions for object-oriented and aspect-oriented refactoring patterns. The approach is exemplified by defining impact functions for an aspect-oriented refactoring pattern, named *Pull Up Advice*, for four metrics: lines of code, number of operations in module, crosscutting degree of an aspect and coupling on advice execution. A case study shows examples of impact functions applied to an open-source application. Tool support helps in creating impact functions both for object-oriented and aspect-oriented refactoring patterns.

7.2 Creating Impact Functions

Impact functions (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006) are mathematical functions that assess the impact of refactoring on software quality by describing the changes in metrics values when a certain refactoring pattern is applied to a software element. Whenever a developer comes upon an opportunity for refactoring and is not sure about the implications of the transformation, an impact function helps to assess the effect of the refactoring.

These impact functions guide the developer in choosing refactoring patterns in each case. The decision of moving features between classes can be supported by the impact functions, which obtain the metric values of each resulting refactoring, without actually performing the transformation. The use of these functions can show which refactoring opportunities are more advantageous (in terms of the metric values). Furthermore, when there are many refactoring opportunities, the developer wants to focus on those that provide the greatest improvements in the software artefacts.

This section describes an approach to create impact functions for object-oriented and aspect-oriented refactoring patterns, including the description of the roles, the activities and the artefacts used in the approach.

7.2.1 Process Roles, Activities and Artefacts

The creation of impact functions is performed once for each pair $\mathcal{P}$ = (refactoring pattern, quality attribute) and is more likely to be performed by a *Quality Analyst*. He performs a set of activities leading to impact functions that evaluate the changes in metric values for a set of refactoring patterns and a set of metrics, as shown in Figure 7.1. Please note that these activities focus only on the creation of impact functions. The use of such impact functions is described in Chapter 3.

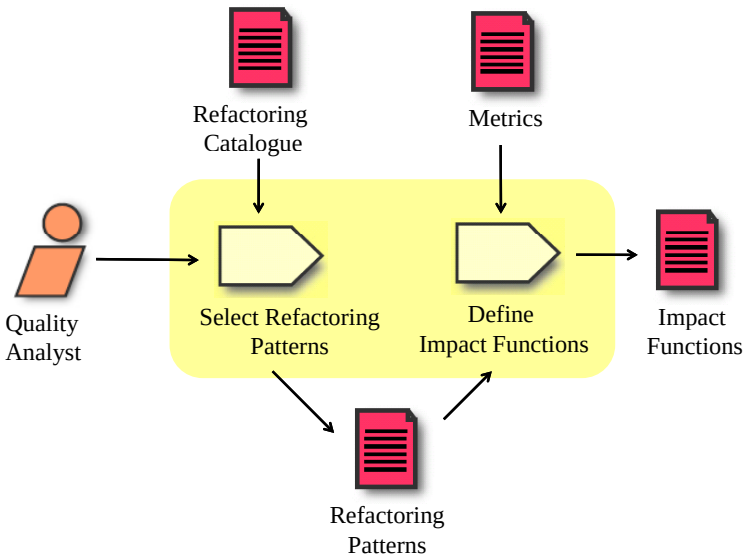

Figure 7.1: Process roles, activities and artefacts: overview

In the first step the quality analyst selects the refactoring patterns for which the impact functions will be created. Refactoring patterns are selected from refactoring catalogues. There are refactoring catalogues for object-oriented software (FOWLER et al., 1999), aspect-oriented software (MONTEIRO; FERNANDES, 2005a, 2006), design models

(MARKOVIC; BAAR, 2005; ZHANG; LIN; GRAY, 2005), and use case models (RUI; REN; BUTLER, 2003; YU; LI; BUTLER, 2004).

In the next step, *define impact functions*, the functions are defined in the form $f(\lambda, \mu, \eta)$, where λ represents a refactoring pattern, μ represents a module and η represents a metric. The function is created to compute the change in the value of the metric η after applying the λ refactoring on module μ, so $f(\lambda, \mu, \eta) = \eta(\lambda(\mu)) - \eta(\mu)$.

An implementation of the refactoring catalogue must be present in the case tool or IDE for automated support for refactoring. If there is no such implementation, the developer can apply refactoring patterns manually, which can be error-prone, or add refactoring capabilities to the tool, which can require a substantial effort.

For each refactoring pattern and each metric, the developer can create impact functions to express how the metric values change when the refactoring pattern is applied to a software element. A refactoring pattern can affect more than one software element. Each affected software element is called a *participant* of the refactoring pattern. If the refactoring pattern has more than one participating software element, the developer must create impact functions for each software element. For example, the *Move Method* refactoring pattern is applied to a method defined in a source class, moving it for a destination class. Both classes are participants of the refactoring pattern.

Let *RPs* be the set of refactoring patterns and *Ms* be the set of metrics for which the impact functions will be created. The algorithm for creating impact functions can be described as follows:

CREATE-IMPACT-FUNCTIONS(RPs, Ms)
1 $F \leftarrow$ NIL;
2 **for each** r **in** RPs
3 **do** $Ps \leftarrow r$.GETPARTICIPANTS()
4 **for each** p **in** Ps
5 **do for each** m **in** Ms
6 **do** $F \leftarrow F \cup \{$CREATE-IMPACT-FUNCTION$(r, p, m)\}$
7
8 **return** F;

The developer creates a set of impact functions, one for each participant and each metric. The call to *getParticipants()* retrieves the set of participating elements of the refactoring pattern, named *Ps*, whereas the call to *Create-Impact-Function(r,p,m)* represents the creation of an impact function for a refactoring pattern *r*, the participating element *p* and a metric *m*.

7.3 Creating Impact Functions for *Pull Up Advice*

This section provides impact functions that compute the changes in metric values when applying the refactoring pattern *Pull Up Advice* onto a set of modules (aspects and classes). Section 7.3.1 describes why the *Pull Up Advice* refactoring pattern was selected. Section 7.3.2 describes the selected metrics: lines of code, number of operations in module, crosscutting degree of an aspect and coupling on advice execution used in the example. Section 7.3.3 shows the application of the approach.

7.3.1 Selecting Refactoring Patterns

The main target of this example focuses on the software elements particular to aspect-oriented languages: pointcuts, advices and inter-type declarations. Given the fact that the refactoring of pointcuts has little effect on software metrics and that inter-type declarations are very similar to their object-oriented counterparts, only one target is chosen for refactoring: advices.

Advices can be moved to other aspects, pulled up to the super-aspect, pushed down to the sub-aspect, converted to a method or inlined into the affected classes or aspects. The *Pull Up Advice* was chosen as a representative refactoring pattern for advices, as it is more common to generalize a behaviour in an advice and moving it to a super-aspect than pushing it down or inlining it.

The *Pull Up Advice* refactoring pattern moves an advice from an aspect to its super-aspect (GARCIA et al., 2004). It is useful when there are duplicated advices in sibling aspects, and when someone wants to move the advice functionalities to the super-aspect in order to better distribute the responsibilities among the super and sub-aspects. In this case, one of the duplicates is chosen to be pulled up, the others can be deleted. *Pull Up Advice* is also used when an aspect has functionalities that can be used (or are effectively being used) by several related aspects, thus reducing code duplication.

7.3.2 Selected Metrics

The developer has to define impact functions for the metrics listed in the quality model(s) defined by the SQA team. The set of metrics used in this example has already been used in a set of experimental studies defined to assess the reusability and maintainability of aspect-oriented software (CACHO et al., 2006; CASTOR FILHO; GARCIA; RUBIRA, 2005; GREENWOOD; BLAIR, 2006).

Some of the metrics (CECCATO; TONELLA, 2004) chosen in this quality model are originally extensions of well-known metrics for object-oriented software (CHIDAMBER; KEMERER, 1994) and are adapted to the context of aspect-oriented software. For a complete empirical study, including empirical data collection, usage scenarios and correlations between those metrics, please refer to Chapter 9. The selected metrics are:

- Lines of Code (locc)
- Number of Operations in Module (nom)
- Crosscutting Degree of an Aspect (cda)
- Coupling on Advice Execution (cae)

7.3.3 Impact Functions for *Pull Up Advice*

This section shows a set of impact functions created for the *Pull Up Advice* refactoring pattern. For all the impact functions, consider an aspect α, an advice ρ, and the set of sub-aspects of α containing ρ, named $\mathcal{B}$. The λ function to pull up the ρ advice from the aspects in the $\mathcal{B}$ set to the α aspect can be specified as:

$$\lambda = pullUpAdvice(\alpha : Aspect, \mathcal{B} : Set\,of\ Aspects, \rho : Advice) \tag{7.1}$$

7.3.3.1 *Impact on locc*

Let μ be an aspect. Let $locc(\mu)$ represents the number of lines of code in this module. The value of the impact functions depends on the size of the pulled up advice:

$$f(\lambda, \alpha, locc) = locc(\rho) \quad (7.2)$$

$$\forall \beta_i \in \mathcal{B}\ f(\lambda, \beta_i, locc) = -locc(\rho) \quad (7.3)$$

Note that each function only shows the difference between the value of the $locc$ metric after and the value of the $locc$ metric before the application of the refactoring pattern.

When applying the *Pull Up Advice* refactoring pattern, the $locc$ metric value depends on the size of the advice being pulled up. The metric value of the super-aspect is raised by the $locc$ value of the advice. The inverse occurs in the affected sub-aspects that do not have the advice anymore, their $locc$ is reduced by the $locc$ value of the advice pulled up. Also, the sum of $locc$ in sub and super aspects only changes when the number of children participating in the refactoring pattern is higher than one. If there is only one child moving the advice to its super aspect, the sum of the $locc$ values remains the same.

When both the number of children and the size of the advice being moved to the super class increase, the benefits of the application of the *Pull Up Advice* refactoring pattern is more evident.

Figure 7.2 shows the mean size of the super and sub-aspects when applying the *Pull Up Advice* refactoring pattern regarding the number of children (from 1 to 5) and regarding the advice size (from 0 to 100 % of the total size of the aspects).

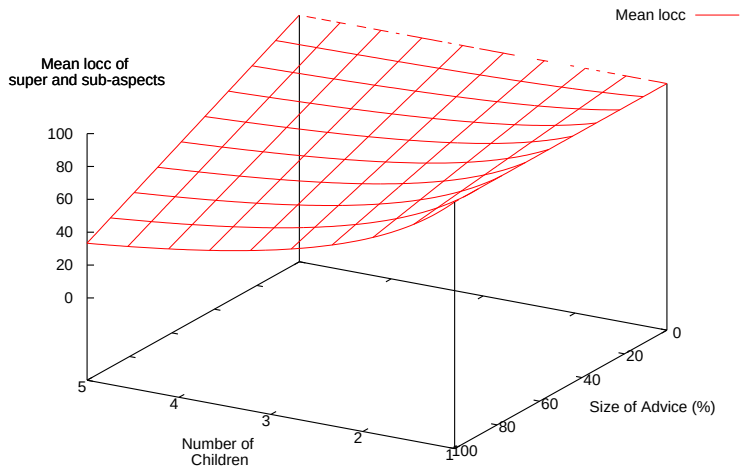

Figure 7.2: Percentage of size decrease per number of children

An initial size of 100 is used for both super and sub-aspects to better visualize the changes in the $locc$ metric value. Note that the decrease in the mean size (measured in $locc$) is directly proportional to the number of children and the advice size. This refactoring pattern is usually applied to remove advice duplication.

7.3.3.2 *Impact on nom*

When applying the *Pull Up Advice* refactoring pattern, the nom metric value of the super-aspect is incremented by one as there is one newly created advice, whilst the values

of the nom of the sub-classes are lowered by one:

$$f(\lambda, \alpha, nom) = +1 \quad (7.4)$$
$$\forall \beta_i \in \mathcal{B}\ f(\lambda, \beta_i, nom) = -1 \quad (7.5)$$

If the language supports the definition of anonymous inner classes then the functions change to the following:

$$f(\lambda, \alpha, nom) = +nom(\rho) \quad (7.6)$$
$$\forall \beta_i \in \mathcal{B}\ f(\lambda, \beta_i, nom) = -nom(\rho) \quad (7.7)$$

Figure 7.3 shows the percentage of size decrease per number of children in the nom metric values regarding the number of children of the super-aspect and regarding the current nom value. The decrease is directly proportional to the number of children and the inversely proportional to the total nom in the super-aspect and its sub-aspects.

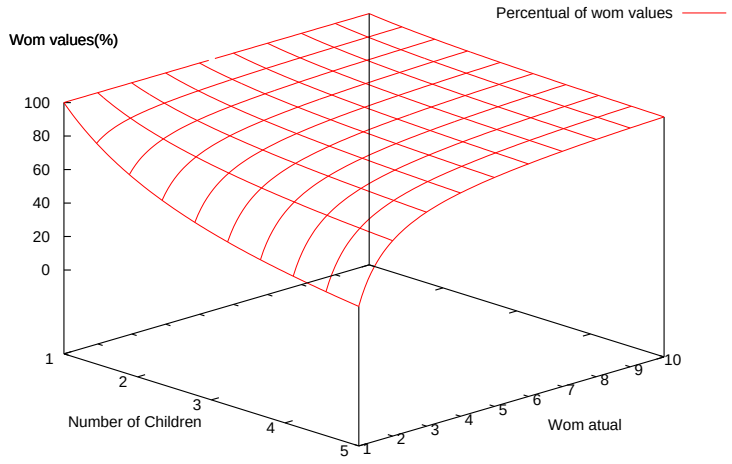

Figure 7.3: Percentage of size decrease per number of children

7.3.3.3 Impact on cda - Changes in metric value for α

Consider $\mathcal{A}_m$ as the set of all advised modules by α and $\mathcal{R}$ the set of advised modules by advice ρ. The change in the value of cda for α after the application of the λ refactoring pattern is given by the cardinality of the set resulting from the difference from $\mathcal{R}$ and $\mathcal{A}_m$: $f(\lambda, \alpha, cda) = |\mathcal{R} - \mathcal{A}_m|$. After being moved to the super-aspect, the advice can advise itself (depending on the pointcut expression). Therefore, the impact function for computing the change in the value of cda of α can be defined by an interval:

$$f(\lambda, \alpha, cda) = [|\mathcal{R} - \mathcal{A}_m|, |\mathcal{R} - \mathcal{A}_m| + 1] \quad (7.8)$$

7.3.3.4 Impact on cda - Changes in metric values for $\mathcal{B}$

Consider a set of sub-aspects $\mathcal{B}$. Let $\mathcal{ME}_i = (\mathcal{E}_i, f_i)$ be the multiset of all modules advised by $\beta_i \in \mathcal{B}$, where $\mathcal{E}_i$ is the set of advised modules and $f : \mathcal{E}_i \rightarrow N$ is a function from $\mathcal{E}_i$ to the set of natural numbers. Let $\mathcal{R}$ be the set of modules advised by ρ advice. The changes in cda for the elements of $\mathcal{B}$ are given by the number of elements that do

not appear in $\mathcal{E}_i$ after removing the affected modules by $\mathcal{R}$ from the multiset $\mathcal{ME}_i$. This number of elements counts the modules that are not affected anymore by the β_i aspect, so the cda metric value is decreased by this number:

$$\forall \beta_i \in \mathcal{B}\ f(\lambda, \beta_i, cda) = -|\mathcal{E}_i - (\mathcal{ME}_i - \mathcal{R})| \tag{7.9}$$

The cda metric value represents the number of modules affected by the pointcuts and by the inter-type declarations in a given aspect. This metric values change according to the modules advised by the moved advice and to the modules already advised by the super-aspect.

Figure 7.4 shows how the cda metric values changes regarding the number of modules affected by the super aspect and regarding the modules affected by advice that are not also affected by the super-aspect. The values for the cda metric are directly proportional to the number of affected modules by the super-aspect and to the different modules affected by the advice.

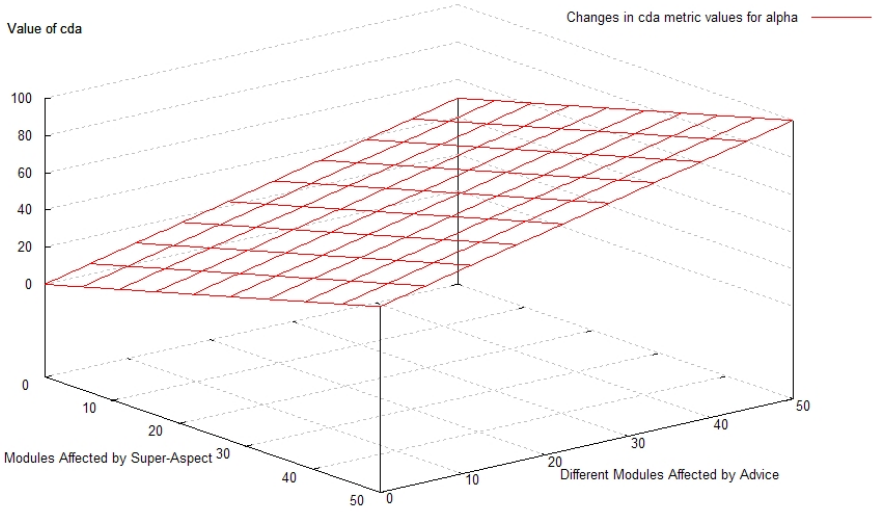

Figure 7.4: Percentage changes in the cda values for the super-aspects

Several different cases might occur when applying *Pull Up Advice*. If the modules affected by the advice are the same affected by the super aspect, the value of the cda metric remains the same. The value does not change also when the modules affected by the advice are a sub-set of the modules affected by the super-aspect. When the modules affected by the advice are different from the ones affected by the super-aspect, the value of cda increases depending on the number of different modules affected. The higher this number of modules, the higher is the change in the cda metric value.

Figure 7.5 shows changes in the cda values for the sub-aspects regarding the modules affected by the sub-aspects and the percentage of modules also affected by advice. The figure shows how the values of cda change when the super-aspect has one, two or three sub-aspects. The value of cda is directly proportional to the modules affected by the sub-aspects and to the percentage of modules also affected by advice.

7.3.3.5 Impact on cae - Changes in the metric value for α

Let $\mathcal{C}$ be the multiset of aspects that advises α (the super-aspect) and $\mathcal{D}$ the multiset of aspects that advises the ρ advice. The value of the impact function considering cae

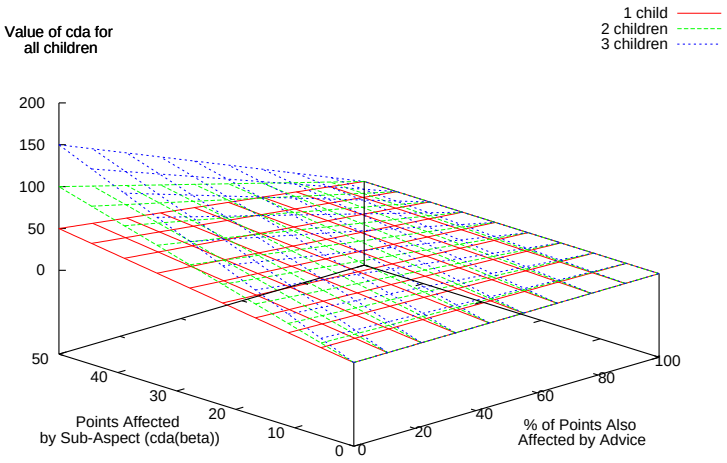

Figure 7.5: Changes in the cda values for the sub-aspects

for the super-aspect α after the application of the λ refactoring pattern is given by the interval that starts in zero and ends in the cardinality of the difference between $\mathcal{D}$ and $\mathcal{C}$. As the aspects that advise $\mathcal{D}$ can change when the advice is moved to the super-aspect, an interval is considered for the value of the impact function:

$$f(\lambda, \alpha, cae) = +[0, |\mathcal{D} - \mathcal{C}|] \tag{7.10}$$

7.3.3.6 Impact on cae - Changes in the metric values for β

The impact in the values of the cae metric in the sub-aspects is the following: Let $\mathcal{B}$ be the set of sub-aspects of a given aspect, $\mathcal{CB}$ the multiset of aspects that advises β and $\mathcal{D}$ the set of aspects that advises the ρ advice. The value of the change in cae for each β_i after the application of the λ refactoring pattern is given by the cardinality of the difference between $\mathcal{CB}$ and $\mathcal{D}$. Thus, the difference between the value before the refactoring process and after is given by the function:

$$\forall \beta_i \in \mathcal{B}\ f(\lambda, \beta_i, cae) = -(|\mathcal{CB}| - |\mathcal{CB} - \mathcal{D}|) \tag{7.11}$$

The cae metric represents the number of aspects that affect a given module. When applying the *Pull Up Advice* refactoring pattern, the values of cae on the super-aspect change according the aspects that affect it and the aspects that affect the advice being pulled up. Figure 7.6 shows changes in the cae values for the super-aspect, regarding the number of aspects that affect the super aspect and the number of aspects that affect only the advice. The maximum and minimum expected values of changes in the cae metric value after the application of the *Pull Up Advice* refactoring pattern are shown.

7.3.3.7 Example

The following example is from the *GlassBox Inspector*. An advice is moved from the *AbstractXmlCallMonitor* aspect to the super-aspect *AbstractXmlProcessingMonitor* to illustrate the use of impact functions for the *Pull Up Advice* refactoring pattern. The advice is defined in lines 3-9. Note that it has an anonymous inner class (lines 4 - 7) that is also counted to obtain the values for the nom metric.

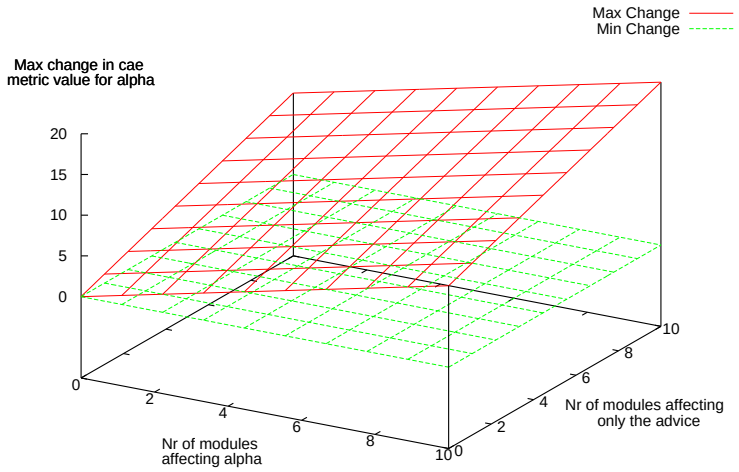

Figure 7.6: Changes in the *cae* values for the super-aspect

```
1   public abstract aspect AbstractXmlCallMonitor
2     extends AbstractXmlProcessingMonitor {
3     protected pointcut scope();
4     Object around(final Node node) :
5       domCall(node) && !inXmlRequest()
6         && monitorEnabled() {
7       RequestContext requestContext =
8         new XmlRequestContext() {
9         public Object doExecute() { ... }
10        public PerfStats lookupStats() { ... }
11      };
12      return requestContext.execute();
13    }
14  }
```

```
1  public abstract aspect AbstractXmlProcessingMonitor
2    extends AbstractResourceMonitor {
3      static Method getDocUriMethod = null;
4      static {...}
5      public pointcut parseCall() :
6          call(public * DocumentBuilder.parse(..));
7      public pointcut domCall(Node node) :
8          call(* org.w3c.dom..*(..)) && target(node);
9      public pointcut saxCall() :
10         call(* org.xml.sax..*(..));
11     protected pointcut inXmlRequestBind
12         (RequestContext context) :
13         inRequest(context) &&
14         if(context instanceof XmlRequestContext);
15     public pointcut inXmlRequest() :
16         inXmlRequestBind(*);
17     protected abstract class XmlRequestContext
```

```
18          extends ResourceRequestContext {...}
19  }
```

The values of the metrics for *AbstractXmlProcessingMonitor* and *AbstractXmlCallMonitor* were collected using the *aopmetrics* tool (STOCHMIALEK, 2009):

$$
\begin{aligned}
\alpha &= AbstractXmlProcessingMonitor \\
locc(\alpha) &= 42 \\
nom(\alpha) &= 1 \\
cda(\alpha) &= 0 \\
cae(\alpha) &= 2 \\
\beta &= AbstractXmlCallMonitor \\
locc(\beta) &= 23 \\
nom(\beta) &= 3 \\
cda(\beta) &= 0 \\
cae(\beta) &= 4
\end{aligned}
$$

The evaluation of the changes in both aspects and classes using the defined impact functions is the following:

$$
\begin{aligned}
\alpha &= AbstractXmlProcessingMonitor, \mathcal{B} = AbstractXmlCallMonitor \\
\rho &= Object\ around(final\ Node\ node), \lambda = pullUpAdvice(\alpha, \mathcal{B}, \rho) \\
locc'(\alpha) &= locc(\alpha) + f(\lambda, \alpha, locc) = 42 + locc(\rho) = 42 + 20 = 62 \\
locc'(\beta) &= locc(\beta) + f(\lambda, \beta, locc) = 23 - locc(\rho) = 23 - 20 = 3 \\
nom'(\alpha) &= nom(\alpha) + f(\lambda, \alpha, nom) = 1 + nom(\rho) = 1 + 3 = 4 \\
nom'(\beta) &= nom(\beta) + f(\lambda, \beta, nom) = 3 - nom(\rho) = 3 - 3 = 0
\end{aligned}
$$

There is the need to collect the aspects that advise α, β and ρ to calculate the values for the impact functions of the *cae* metric. The multiset of aspects that advises α, β and ρ are named $\mathcal{C}$, $\mathcal{CB}$, $\mathcal{D}$, respectively.

$$
\begin{aligned}
\mathcal{C} &= \{(JmxManagement, 1), (SimpleConfig, 1)\} \\
\mathcal{CB} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 2), (TrackParents, 1)\} \\
\mathcal{D} &= \{(ErrorHandling, 2), (TrackParents, 1)\} \\
cae'(\alpha) &= cae(\alpha) + f(\lambda, \alpha, cae) = 2 + [0, |\mathcal{D} - \mathcal{C}|] = 2 + [0, 2] = [2, 4] \\
cae'(\beta) &= cae(\beta) + f(\lambda, \beta, cae) = 4 + (-|\mathcal{CB}| - |\mathcal{CB} - \mathcal{D}|) = 4 - 2 = 2
\end{aligned}
$$

The super and sub-aspect do not affect any modules, so the set of modules $\mathcal{A}_d, \mathcal{R}$ and $\mathcal{ME}$ are empty:

$$
\begin{aligned}
\mathcal{A} &= \{\}, \mathcal{R} = \{\}, \mathcal{ME} = \{\}, \\
cda'(\alpha) &= cda(\alpha) + f(\lambda, \alpha, cda) = 0 + [0, 1] = 1 \\
cda'(\beta) &= cda(\beta) + f(\lambda, \beta, cda) = 0 + 0 = 0
\end{aligned}
$$

Note that the advice, having moved to the super-aspect now affects itself and, therefore, the value of the cda metric is increased by one. The new metric values of the *AbstractXmlProcessingMonitor* and *AbstractXmlCallMonitor* aspects computed with the help of impact functions, without the application of the *Pull Up Advice* refactoring pattern are the following:

$$\begin{aligned}
\alpha &= AbstractXmlProcessingMonitor \\
locc(\alpha) &= 62 \\
nom(\alpha) &= 4 \\
cda(\alpha) &= 1 \\
cae(\alpha) &= 4 \\
\beta &= AbstractXmlCallMonitor \\
locc(\beta) &= 3 \\
nom(\beta) &= 0 \\
cda(\beta) &= 0 \\
cae(\beta) &= 2
\end{aligned}$$

Now that the advice is in the *AbstractXmlProcessingMonitor* aspect (line 2), the *AbstractXmlCallMonitor* aspect (lines 4-6) can be removed, as it only contains a pointcut that is not referenced by any module.

Note that, in some cases, the pointcut expressions are only evaluated in runtime, and the computed results using the impact function are restricted to the types of pointcuts that can be evaluated at weaving time. In these cases, the impact functions provide *estimated* values about the effects of refactoring in software metrics.

7.4 Case Study: Computing the Values of *Pull Up Advice* in the Glassbox Inspector

This second case study aims at computing the values of impact functions for all the aspects in the Glassbox Inspector in which it is possible to apply the *Pull Up Advice* refactoring pattern.

For all the impact functions to be computed, consider super-aspect α, a sub-aspect $\mathcal{B}$ and an advice ρ. The λ function to pull up an advice can be specified as:

$$\lambda = pullUpAdvice(\alpha : SuperAspect, \beta : SubAspect, \rho : Advice) \tag{7.12}$$

Three aspects (the α aspects) have sub-aspects, as follows:

$$\begin{aligned}
\alpha_1 &= AbstractResourceMonitor \\
\alpha_2 &= AbstractXMLProcessingMonitor \\
\alpha_3 &= AbstractRequestMonitor
\end{aligned}$$

Six aspects are the sub-aspects (the β aspects):

$$\begin{aligned}
\beta_1 &= JDBCConnectionMonitor \\
\beta_2 &= JDBCStatementMonitor \\
\beta_3 &= RemoteCallMonitor \\
\beta_4 &= JaxmCallMonitor \\
\beta_5 &= AbstractXMLCallMonitor \\
\beta_6 &= AbstractOperationMonitor
\end{aligned}$$

In the β aspects, there are 15 different advices which can be pulled up to the super-aspects:

$$\begin{aligned}
\rho_1 &= around(DataSource) \\
\rho_2 &= around(String) \\
\rho_3 &= before(Statement, String) \\
\rho_4 &= around(Statement) \\
\rho_5 &= afterreturning(Connection) \\
\rho_6 &= around(String) \\
\rho_7 &= around(Object) : remote \ldots \\
\rho_8 &= around(Object) : jaxRPC \ldots \\
\rho_9 &= around(Object, Object, Object) \\
\rho_{10} &= around(Node) \\
\rho_{11} &= afterreturning(Object) \\
\rho_{12} &= around(Object) : class \ldots \\
\rho_{13} &= around(Object) : methodSig \ldots \\
\rho_{14} &= around(Object) : methodNameCon \ldots \\
\rho_{15} &= around(Object) : methodCon \ldots
\end{aligned}$$

Table 7.1 shows the relationship between the super-aspect, the sub-aspects and the advices. Each application of the *Pull Up Advice* refactoring pattern is represented by a numbered λ. For example, the λ_1 application moves the ρ_1 advice (*around(DataSource)*) from the β_1 sub-aspect (*JDBCConnectionMonitor*) to the α_1 super-aspect (*AbstractResourceMonitor*).

The detailed computation of each metric value for both the super aspects and sub-aspects, and the source code and metrics for each advice are shown in Appendix E. The computation of the impact functions for the proposed application of the *Pull Up Advice* refactoring pattern leads to two different result sets: the impact function values for the super-aspects and the values for the sub-aspects.

Table 7.2 shows the values for the super-aspects. The impact functions provide to the developer information of how much each application of a specific refactoring pattern affects the values of a set of chosen metrics. The detailed computation of each value for the impact functions of this case study is shown in Appendix E.

For example, if the goal of the developer is to increase the size of the super-aspect, the best application of the *Pull Up Advice* refactoring pattern is λ_{10}, which increases the $locc$

Table 7.1: Relationship between the super-aspect, the sub-aspects and the advices.

λ	α	β	ρ
λ_1	α_1	β_1	ρ_1
λ_2	α_1	β_1	ρ_2
λ_3	α_1	β_2	ρ_3
λ_4	α_1	β_2	ρ_4
λ_5	α_1	β_2	ρ_5
λ_6	α_1	β_2	ρ_6
λ_7	α_1	β_3	ρ_7
λ_8	α_1	β_3	ρ_8
λ_9	α_1	β_4	ρ_9
λ_{10}	α_2	β_5	ρ_{10}
λ_{11}	α_3	β_6	ρ_{11}
λ_{12}	α_3	β_6	ρ_{12}
λ_{13}	α_3	β_6	ρ_{13}
λ_{14}	α_3	β_6	ρ_{14}
λ_{15}	α_3	β_6	ρ_{15}

Table 7.2: The values for the computation of impact functions for *Pull Up Advice*, considering the super-aspect.

λ	**f(λ, α, locc)**	**f(λ, α, nom)**	**f(λ, α, cda)**	**f(λ, α, cae)**
λ_1	+10	+2	+[0,1]	+[0,1]
λ_2	+10	+2	+[0,1]	+[0,1]
λ_3	+4	+1	+[0,1]	0
λ_4	+10	+2	+[0,1]	+1
λ_5	+6	+1	+[0,1]	0
λ_6	+11	+3	+[0,1]	+[0,1]
λ_7	+13	+3	+[0,1]	+[0,1]
λ_8	+13	+3	+[0,1]	+[0,1]
λ_9	+11	+3	+[0,1]	+[0,1]
λ_{10}	+17	+3	+[0,1]	+[0,1]
λ_{11}	+3	+1	+[1,2]	0
λ_{12}	+11	+3	+[0,1]	0
λ_{13}	+11	+3	+[0,1]	0
λ_{14}	+11	+3	+[0,1]	0
λ_{15}	+11	+3	+[0,1]	0

of α by 17 lines of code. If the goal is to decrease the super-aspect, then λ_3 or λ_{11} are the best choices in terms of $locc$. The same kind of analysis can be performed for the other metrics, depending on the goals of the developer.

Table 7.3 shows the values for the sub-aspects. For example, if the developer is looking to decrease the interaction of aspects, he can focus to decrease the value of the cae metric, which represents the number of aspects affecting this specific module. In this case, the refactoring pattern applications of choice are λ_7, λ_8, λ_9 or λ_{10}.

If heuristic rules are being used to prioritise refactoring opportunities, the developer can compose the impact functions to compute the changes in the values of those rules.

Table 7.3: The values for the computation of impact functions for *Pull Up Advice*, considering the super-aspect.

λ	**f(λ, β, locc)**	**f(λ, β, nom)**	**f(λ, β, cda)**	**f(λ, β, cae)**
λ_1	-10	-2	0	0
λ_2	-10	-2	0	0
λ_3	-4	-1	0	0
λ_4	-10	-2	0	0
λ_5	-6	-1	0	0
λ_6	-11	-3	0	0
λ_7	-13	-3	0	-1
λ_8	-13	-3	0	-1
λ_9	-11	-3	0	-2
λ_{10}	-17	-3	0	-2
λ_{11}	-3	-1	-1	0
λ_{12}	-11	-3	0	0
λ_{13}	-11	-3	0	0
λ_{14}	-11	-3	0	0
λ_{15}	-11	-3	0	0

7.5 Tool Support: An API for the Creation of Impact Function

Tool support for using impact functions is desirable to provide mechanisms to create and compute impact functions. To allow that, it must be possible for the developer to:

1. **Manipulate metrics.** The tool must provide support for the creation of functions to compute metrics or it must provide integration with external tools for computing the metric values of software elements.

2. **Manipulate refactoring patterns.** The tool must provide support for cataloguing refactoring patterns or must provide integration with existing cataloguing tools.

3. **Create impact functions.** The tool must provide support for the creation of impact functions. These functions associate metrics with refactoring patterns, and therefore, the creation of impact functions depends on the previously described features (manipulation of metrics and refactoring patterns).

4. **Compute impact functions.** Tool support must be provided for the correct computation of the impact functions defined.

In this work, a proof-of-concept API was developed to automatically evaluate changes in metric values. The developer can use the existent impact functions associated with a set of refactoring patterns or can create new ones.

Each refactoring pattern has a set of participants (source aspect and destination aspect, for example) and can be associated with a set of metrics. For each pair $\mathcal{P}_m =$ (participant, metric), the developer can inform a different impact function. The developer can also associate more than one metric and impact function in a single operation.

The classes representing refactoring patterns are instantiated and the impact functions can be computed for actual participants, provided by the user. This computation returns the modified metric values. Refactoring in sequence can be performed by sequentially computing individual impact functions.

Figure 7.7 shows the main packages of the API developed to create and compute impact functions. The *refactoring* package provides main support for the specification of refactoring patterns. The *model* package provides classes to deal with metrics (in general) and wrappers to make easy to work with specific metrics. The *impactFunction* package provides classes to create impact functions using both refactoring patterns and metrics. The tests package can provide test cases for the created impact functions.

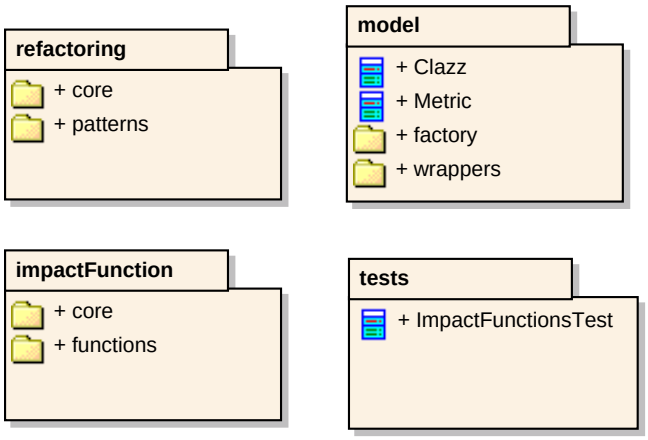

Figure 7.7: Impact functions API - packages

Figure 7.8 shows the main classes dealing with impact functions. The *Refactoring* class represents refactoring patterns and includes support for adding refactoring participants (the software elements which are the target for refactoring), adding impact functions and computing impact functions. Subclasses are provided to represent refactoring patterns targeted to a single class or aspect (such as *DeleteAttribute* and *ExtractPointcut*) and targeted to a pair of classes (such as *Move Method*).

Future work can focus on expanding the existing refactoring patterns and impact functions, providing integration with metric collectors for object-oriented and aspect-oriented software, commercial IDEs and refactoring tools.

7.6 Related Work

Mens et al. (MENS et al., 2003) state that an open problem is to assess the effects of a refactoring pattern on the software quality, as some refactoring patterns remove redundancy, raise abstraction or modularity level and others have negative impact on reusability, for example. They also observe that determining where and why refactoring patterns should be applied is still an open problem. By classifying refactoring patterns in terms of the quality attributes they affect, the effect of a refactoring on the software quality can be estimated.

This thesis provides a quantitative approach to determine advantageous opportunities for refactoring, considering the selected quality attributes. The approach is based on the use of metric and heuristic-based functions to spot shortcomings in software elements that can be improved through refactoring. The identified opportunities for refactoring can then be prioritised and the effects on software quality of each suggested refactoring can be assessed using impact functions and heuristic rules. DuBois and Mens (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005) propose a formalism to describe the impact of a representative number of refactoring patterns on an AST representation of the source

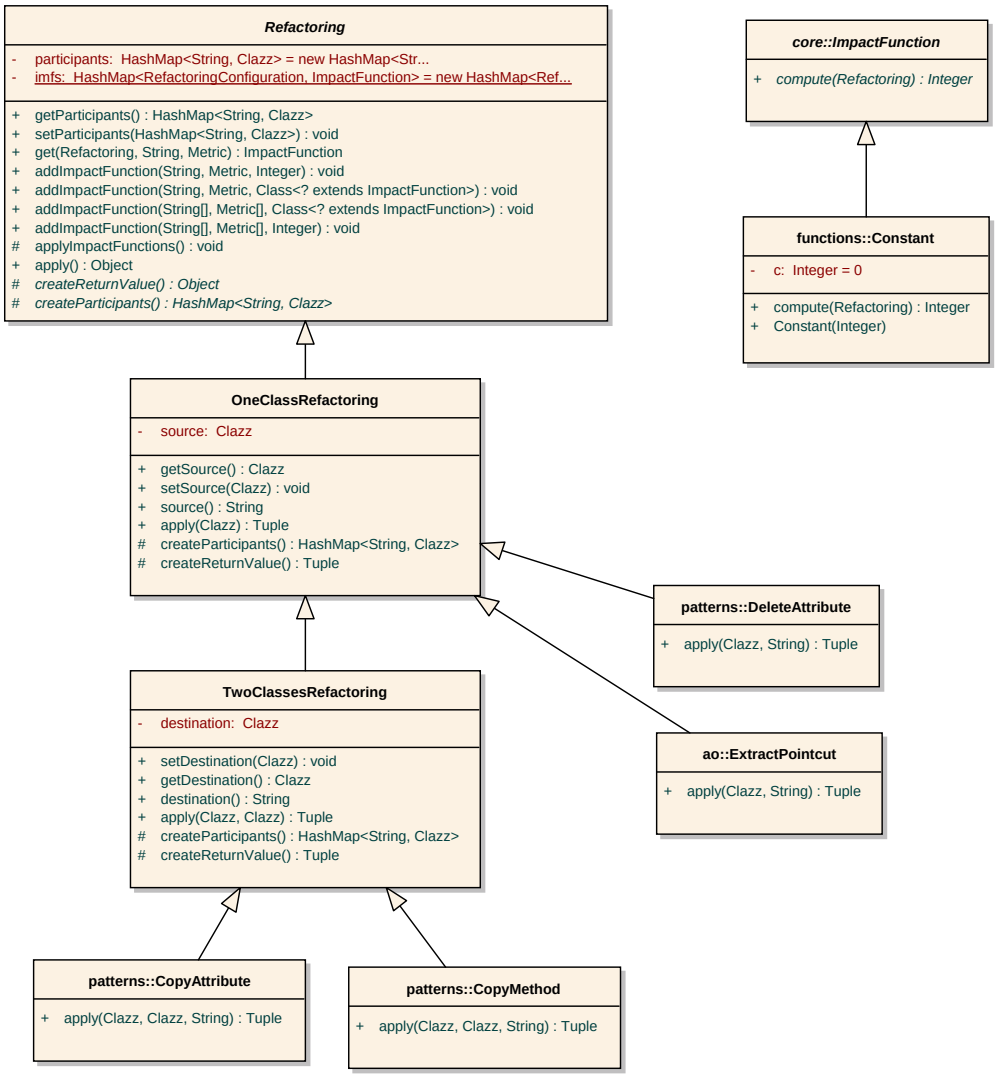

Figure 7.8: Impact functions API - classes

code, extended with cross-references. The proposed formalism uses AST representing object-oriented programs and metrics. They evaluate the impact of *Extract Method*, *Encapsulate Field* and *Pull Up Method* refactoring patterns and describe how quality metrics can be defined on top of this program structure representation.

Du Bois and Mens (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005) and Du Bois (BOIS, 2006) demonstrate formal analysis of refactoring patterns and indicative coupling and cohesion metrics to identify the conditions under which the application of these refactoring patterns minimize coupling and maximize cohesion. They formally define a set of prediction functions to evaluate the impact of the *Extract Method*, *Move Method* and *Replace Method with Method Object* refactoring patterns.

Both works (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006) focus on functions dealing with object-oriented transformations. This chapter deals with refactoring patterns for aspect-oriented software, including a detailed approach for the creation of impact functions. Their formal analysis can be used together with the approach of this thesis to provide additional information for the developer to express the relative importance of refactoring patterns over the quality attributes using pairwise com-

parisons. This thesis proposes an approach to identify refactoring opportunities in software elements using impact functions and their impact functions can serve as a starting point to the definition of more complex functions. The impact functions can be also used together with the approach of this chapter to compute the metric values without the need to apply each refactoring pattern to evaluate their effects.

7.7 Conclusions

By evaluating the impact of refactoring patterns in a software element, the developer observes the changes in metric values and can take an informed decision on whether applying the refactoring patterns improves the quality of the software application.

This chapter proposed an approach to create impact functions for refactoring patterns for aspect-oriented software. The use of these functions to predict the impact of refactoring patterns on software quality can lead to better decisions while modifying existing aspect-oriented software applications. The approach is exemplified using source code metrics. Nothing prevents it, however, to be used to evaluate the effects of refactoring in analysis or design models, for example. Further investigation can be carried to evaluate the effectiveness of these impact functions for assessing the effects of refactoring in software models.

Although the relation of metrics and quality attributes is not discussed in this chapter, the computed metric values can be used to quantitatively assess a set of quality attributes in an application.

The developed tool support provides a starting point to the creation of a more complete catalogue of impact functions, adapting the impact functions described in this chapter for other refactoring patterns, as well as providing mechanisms to define impact functions for object-oriented languages.

Future work should focus on applying the same approach to other aspect-oriented and object-oriented refactoring patterns and focusing on different metrics.

8 REFACTORING SEQUENCES SIMPLIFICATION

This chapter proposes an approach to narrow the number of possible refactoring sequences to be evaluated for a software application. It is organised as follows. Section 8.1 describes the benefits of having an approach to reduce the number of refactoring sequences. Section 8.2 presents the motivation such reduction. Section 8.3 describes how to create an initial representation of the possible sequences and how to reduce the size of this representation. Section 8.4 shows how the approach can be used in practice using a set of refactoring patterns to manipulate methods. Section 8.5 describes tool support and Section 8.6 describes related work. Section 8.7 presents concluding remarks and directions for future work.

8.1 Introduction

Refactoring patterns are usually low grained transformations and the application of an individual refactoring pattern may not bring substantial benefits to the software quality. Current research on the identification of refactoring opportunities (BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; BOIS, 2006) focuses on improvements considering the individual application of a refactoring pattern, but it does not consider how this identification can be conducted in the context of sequences of refactoring patterns.

Even when only a few refactoring patterns are being considered, the number of possible sequences is very large. Consequently, it is important to answer the following questions:

(a) How can the refactoring sequences be narrowed to those that are possible and avoiding sequences which lead to the same results?

(b) And then, from these remaining sequences, how can the developer know which are the best sequences (according to the quality attributes)?

Considering the search for opportunities of refactoring sequences, the main problem is the size of the search space (there are too many possible sequences to be evaluated). This chapter proposes an approach to narrow the number of refactoring sequences by discarding those that does not make sense and avoiding those that lead to the same results.

A detailed example of the approach is provided, considering sequences for method manipulation, showing how the number of sequences can be significantly reduced. It uses deterministic finite automata (DFA) (SIPSER, 1996) to represent refactoring sequences for method manipulation and a set of rules to reduce the search space. It is shown that the number of sequences can be greatly reduced. In the example, the search space showed a

62% reduction on the number of sequences, and in the evaluated projects the reduction is between 57-60%.

8.2 Motivation

The number of software elements where it is possible to apply refactoring patterns can be large. Therefore, when searching for refactoring opportunities, it can be convenient to reduce the search space either by reducing the number of refactoring patterns in a catalogue (by using a pre-defined criteria, such as its position in a ranking of refactoring patterns, as described in Chapter 4) or by imposing constraints for each refactoring pattern (i.e. "opportunities to apply *Pull Up Method* will only consider the immediate superclass as the parameter representing the destination of the method"). Consider, for example, the possible software elements for four different refactoring patterns: *Pull Up Method*, *Extract Interface*, *Rename Class* and *Inline Method*. The number of targets for refactoring for a given set of classes can be counted as:

- **Pull Up Method:** For each class that has a super-class, all the methods can be targets. Let C_c be the set of classes with a super-class and let $m(x)$ be a function that, given a set of classes returns a set of methods defined on these classes, the number of possible applications of this refactoring pattern is $|m(C_c)|$.

- **Extract Interface:** For each class, several different interfaces can be created: an empty interface, an interface with one method (any method of the class), or with two or more methods. Considering $\mathcal{C}$ as the set of classes in a given system, and $n(c)$ as the number of methods of a class c, the number of possible applications of this refactoring pattern in a class c can be computed by the $ei(c)$ function:

$$ei(c) = \sum_{i=0}^{n(c)} \frac{n(c)!}{i!(n(c)-i)!} \tag{8.1}$$

- **Rename Class:** All classes can be renamed so, there is one target per class. In this case, the number of possible applications is $|C|$, where C is the set of classes in a given system. Note that, in this case, the variability in the parameter values (i.e. the new name) is not taken into account.

- **Inline Method:** For each method, in each class, there is one possible target for refactoring. Using the $m(x)$ function, the number of possible applications of this pattern is given by $|m(C)|$.

The number of possible applications of the selected refactoring patterns can be computed by the following equation:

$$P \;=\; |m(C_c)| + \sum_{i=1}^{|C|} ei(c_i) + |C| + |m(C)|$$

Considering a simple system, with ten classes (c_1, ..., c_{10}), with ten methods in each class, where the first five classes are sub-classes, the number of possible refactoring ap-

plication is:

$$\begin{aligned} C &= \{c_1 \dots c_{10}\} \\ Cc &= \{c_1 \dots c_5\} \\ P &= m(C_c) + \sum_{i=1}^{|C|} ei(c_i) + \\ & \quad 5 + m(C) \\ P &= 50 + 1024 + 5 + 100 = 1179 \end{aligned}$$

This example shows that, even for a small software system, the number of possible software elements to apply refactoring patterns is large. Considering the application of refactoring sequences, the number can greatly increase, as for each individual application, the developer can search for additional refactoring patterns that can be applied to the software.

A combination of these sequences is computed by P:

$$P = \binom{n}{k} \tag{8.2}$$

where n is the number of possible applications of the refactoring patterns and k is the sequence size.

In the example, it is impractical to evaluate all possible sequences ($\binom{1179}{k}$). There must be additional constraints defined to further reduce the search space. The set of refactoring opportunities can be reduced using two different strategies:

a **Reducing the number of possible combinations of the refactoring patterns.** This is the focus of this chapter. Section 8.3 describes an approach to reduce the number of possible refactoring sequences and Section 8.4 exemplifies the approach for a set of refactoring patterns.

b **Reducing the parameters to be passed to the refactoring patterns.** For example, the developer can search for refactoring opportunities of the *Extract Interface* refactoring pattern with zero, 50% and 100% of the methods, to reduce the total number of targets (in this case, reducing from 1024 to 254 opportunities). This strategy is not addressed in this thesis and is the focus of future research.

8.3 Reducing the Search Space

This section describes the steps needed to reduce the number of refactoring sequences to be evaluated by the developer. To achieve that, two activities must be performed:

- **Create the initial refactoring sequences:** An initial representation of all the possible sequences of refactoring is created, regardless of the semantics of each refactoring pattern. A representation of refactoring sequences is the use of a notation to express the ordering of application of refactoring patterns. It is used in this approach to express which are the initial possible sequences, by adding sequences to the representation and to reduce the number of sequences, by removing sequences from the representation. The refactoring sequences can be expressed using different representations, such as: trees, graphs, finite state machines, Petri nets, DFAs, and grammars, for example. This representation is manipulated to insert, remove and search for possible refactoring sequences.

- **Simplify the set of sequences:** The initial representation is simplified, considering the semantics of each transformation. In this step, the created representation is traversed, searching for simplifications or equivalences between different sequences. The possible types of simplifications are discussed in Section 8.3.2.

Figure 8.1 shows the roles and artefacts for these two activities. The tool provider creates the initial representation for the refactoring sequences and then simplifies it using a set of rules. A developer can then use this simplified representation to search for refactoring opportunities for these sequences (or for a sub-set of these sequences).

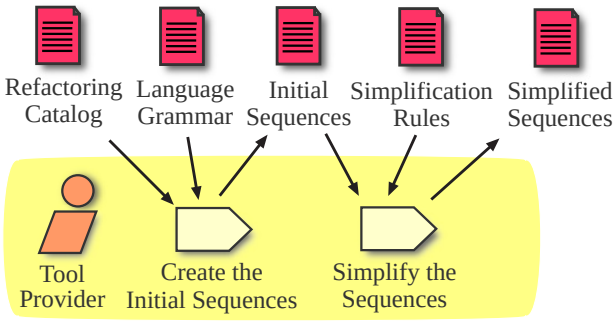

Figure 8.1: Roles, artefacts and activities

Section 8.3.1 describes the creation of this initial representation using a non-deterministic finite automaton and Section 8.3.2 describes the simplification rules and how each simplification can be done.

8.3.1 Creating the Initial Refactoring Sequences

A refactoring pattern is applicable to one or more symbols of a grammar (either terminals or non-terminals) and vice-versa (in this case, each symbol can have *n* applicable refactoring patterns associated to it). A grammar is usually represented by a 4-tuple (N, $\sum$, P, S), in which: N is the finite set of non-terminal symbols, $\sum$ is the finite set of terminal symbols, S is the initial non-terminal symbol and P is a finite set of production rules.

Refactoring sequences are composed of two or more refactoring patterns which are applied in sequence. To create the initial sequences, there is the need to know for which set of refactoring patterns the sequences will be created. For the sake of simplicity, it is suggested to group the refactoring patterns by the grammar symbols they affect. For instance, the sequences in the example are sequences for the manipulation of methods, which affects the non-terminal *method* of an object-oriented language.

To create the initial refactoring sequences, there is the need to bind the set of refactoring patterns to the grammar of the language for which the sequences will be created. This thesis uses the term *refactoring catalogue* to denote a named set of refactoring patterns. Figure 8.2 shows the encapsulation of the binding concern in a separated class. The advantage of separating the binding concern this way is that the catalogue does not need to deal with language-specific grammars, neither the grammars need to be aware of the existence of refactoring patterns that can affect programs written according to its production rules.

When computing the possible sequences, there is the need to inform the maximum size of a sequence. This size is called the *levels of a refactoring sequence*. For example,

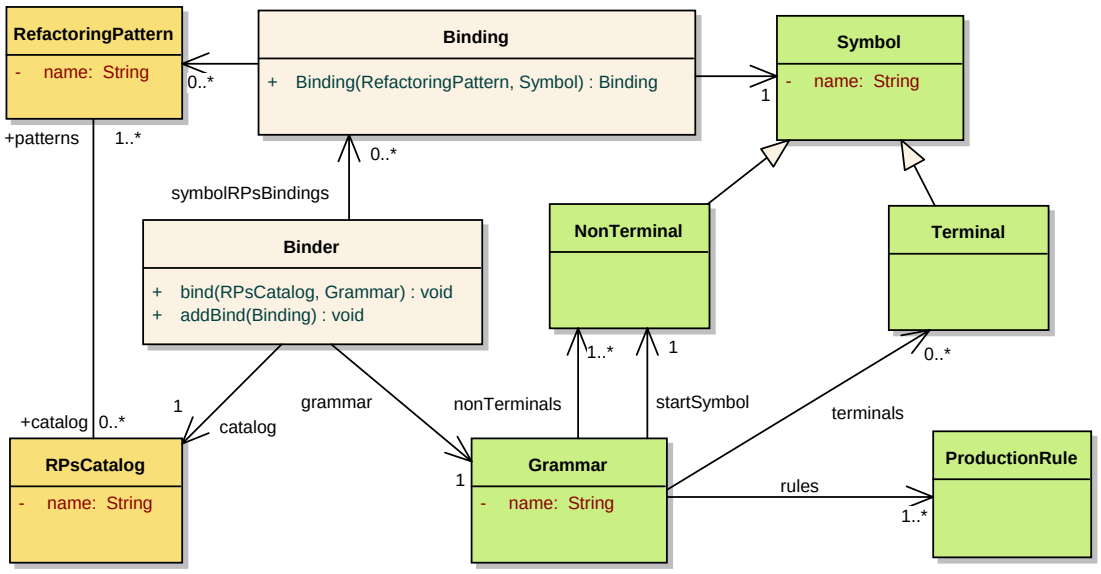

Figure 8.2: Binding refactoring patterns to grammar symbols

the application of a *Pull Up Method* refactoring pattern followed by an application of the *Inline Method* refactoring pattern is a refactoring sequence with two levels. The first one is comprised by the application of *Pull Up Method* and the second one by the application of *Inline Method*.

Considering the existence of a grammar and a refactoring catalogue, the steps for the creation of the initial refactoring sequences for n levels are:

1. Create the binding between the grammar and the refactoring catalogue.

2. Create the individual bindings between the grammar symbols and the individual refactoring patterns in the catalogue.

3. For each grammar symbol which has applicable refactoring patterns proceed as follows:

   (a) Generate all the combinations (with repetitions) of the applicable refactoring patterns with 1 level

   (b) ...

   (c) Generate all the combinations (with repetitions) of the applicable refactoring patterns with n levels

Section 8.4 exemplifies how these steps are mapped to the source code of a prototype developed to generate these initial sequences. To simplify the creation process, the impossible sequences are not evaluated at this stage, but later, in the simplification phase.

8.3.2 Simplifying the Sequences

The next steps reduce the number of sequences. Let $r1$ and $r2$ be refactoring patterns and $\mathcal{I}$ be an initial program which will be manipulated by the refactoring patterns. The following cases can occur:

- **Simplifications:** Simplifications occur when there is a shorter path that leads from an initial program to an end program. If from $\mathcal{I}$, the application of $r1$ followed by $r2$ results in the same piece of software than the application of $r2$ from the initial program $\mathcal{I}$, the sequences are said to be equivalent. This equivalence can be denoted as: $\mathcal{I}\ r1\ r2 \equiv \mathcal{I}\ r2$.

- **Commutative Path:** Commutative paths occur when the order of application of a refactoring patterns pair does not matter. It means that: $\mathcal{I}\ r1\ r2 \equiv \mathcal{I}\ r2\ r1$.

- **Inverse Path:** Refactoring patterns usually have an inverse refactoring pattern (for example, *Pull Up Method* is the inverse refactoring pattern of *Push Down Method*). This case can be expressed as: $\mathcal{I}\ r1\ r2 \equiv \mathcal{I}$.

- **Independent Path**: This kind of sequence occurs when two different refactoring patterns in a path do not have an influence on each other and can be applied in parallel. It is a special case of a commutative path. This cases occur because the refactoring patterns are manipulating distinct elements of the programs and in the general case of commutative paths the refactoring patterns can be manipulating the same elements, but with the same final result.

- **Impossible Paths**: Impossible paths are refactoring sequences that cannot be applied. Certain refactoring patterns can disable the application of other patterns. For example, after a method is inlined it cannot be moved or renamed because the method itself does not exist anymore.

These rules of behavioural preservation and equivalence can be proved using different techniques. The equivalences for simplifications, commutative and inverse paths can be proved using graph parallelism and confluence techniques (BALDAN et al., 1999; HECKEL; KUSTER; TAENTZER, 2002). The occurrence of independent and impossible paths can be detected using critical pair analysis (MENS; TAENTZER; RUNGE, 2005).

The following algorithm can be used to simplify a set of sequences for a given element of the grammar. First, all possible sequences are computed. Then, the impossible, independent and inverse sequences are removed, simplifications are removed and finally, one of the commutative sequences are also removed (it does not matter which one). The algorithm can be expressed as:

SIMPLIFY-REP(rep)
1 $seqs = \text{GETSEQS}(rep)$
2 $seqs = seqs - \text{IMPOSSIBLESEQ(SEQS)}$
3 $seqs = seqs - \text{INDEPENDENTSEQ(SEQS)}$
4 $seqs = seqs - \text{INVERSESEQ(SEQS)}$
5 $seqs = seqs - \text{SIMPLIFICATIONSEQ(SEQS)}$
6 $seqs = seqs - \text{COMMUTATIVESEQ(SEQS)}$
7 **return** $seqs$

Functions *ForbiddenSeq*, *IndependentSeq* , *InverseSeq*, *SimplificationSeq* and *CommutativeSeq* return, respectively, all the sequences that are impossible, independent, inverse, simplifications and one of two commutative paths. An example using these rules is shown in Section 8.4.

These simplification rules can be created once and stored in a knowledge base. They are used when the developer wants to search for sequences of refactoring patterns (instead of the application a single refactoring pattern). A tool provider, for example, can specify a set of simplification rules for refactoring patterns manipulating methods, and another set for refactoring patterns manipulating classes.

The user can then use such rules indirectly, by providing for which refactoring patterns he wants to search for refactoring opportunities, how many levels are the sequences and for which modules, packages or classes the search will be conducted (scope reduction).

8.4 Case Study: Reducing Sequences of Refactoring Patterns for Methods

In this section, deterministic finite automata (DFAs) are used to represent the possible refactoring sequences for a set of refactoring patterns. DFAs are a practical model of computation (SIPSER, 1996), and there are several ways to find a DFA recognizing the union, intersection, and complements of languages.

An DFA is used to exemplify the approach. It is simplified using the previously defined rules. The simplification of the DFA for the *Pull Up Method* paths is explained in detail and for the other remaining paths is briefly described. The initial and final DFAs are compared, showing the differences before and after the simplification is performed.

Table 8.1 shows five refactoring patterns for manipulating methods in a class, as composing the refactoring patterns catalogue, for the purpose of this example.

Table 8.1: Selected refactoring patterns

Ref. Pattern	**Description**
Pull Up Method	The method is moved to one of its superclass (either the immediate one or other super-classes higher in the class hierarchy). This refactoring pattern receives two parameters: a reference to the method to be moved and the super-class of destination.
Inline Method	All calls to the method are replaced by the contents of the method. The method is deleted. This refactoring pattern receives a reference to the method to be inlined as a parameter.
Rename Method	The method is renamed as well as all calls to it. It receives a reference to the method and the new name as parameters.
Move Method	The method is moved to another class and the references are updated. The refactoring pattern receives a reference to the method and the destination class as parameters.
Push Down Method	The method is moved to one or more subclasses of the class containing the method. The refactoring pattern receives a reference to the method and a set of subclasses of destination

8.4.1 Creating the Initial DFA and Removing Impossible Sequences

The first step is to create a DFA describing the sequences of refactoring patterns for method manipulation. The following representations are used to express each refactoring pattern:

- pu(m,s) = pullUp (method, superClass)
- im(m) = inline (method)
- rm(m,n) = rename (method, newName)
- mm(m,nc) = move (method, newClass)
- pd(m,sc) = pushDown (method, subClasses)

As observed, the Inline Method refactoring pattern stops the possibility of manipulating the method. Therefore, the first step is to remove the impossible paths thus generated. Figure 8.3 shows the initial DFA with the impossible paths removed.

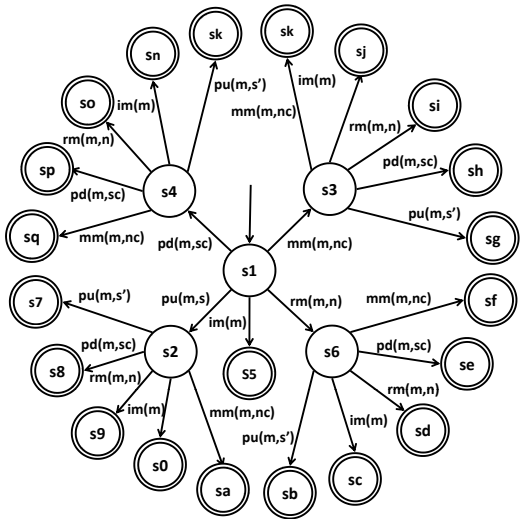

Figure 8.3: Refactoring methods initial DFA, with 21 paths

Note that even though the refactoring patterns *can* be applied, this does not imply that it is sound or beneficial to apply them. In the case of impossible paths, which are sequences that are always impossible, no matter which parameters are used, they are removed *a priori* before the actual applications of each sequence are evaluated in terms of the preconditions of the refactoring patterns. Excluding those impossible paths, from the initial state $s1$, 21 remaining different paths can be followed.

8.4.2 Simplifying the *Pull Up Method* Sequences

In the second step, after the application of a *Pull Up Method* refactoring pattern, other patterns can be applied in sequence, such as: another *Pull Up Method*, an *Inline Method*, a *Rename Method*, a *Move Method* or a *Push Down Method*. The process starts with this particular branch of the initial DFA (Figure 8.3). Figure 8.4 shows the branch of *Pull Up Method* considering the application of two patterns in sequence. This *Pull Up Method* branch will be simplified by applying a set of rules.

The following subsections discuss a set of simplifications made to this branch and then the resulting final DFA is compared with the initial one.

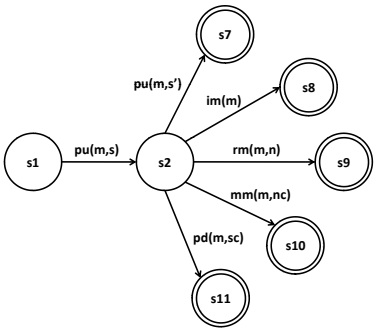

Figure 8.4: *Pull Up Method* - Level 2

8.4.2.1 Simplification: Pull Up Method – Pull Up Method Sequences.

The DFA is traversed, starting from the first sequence (*Pull Up Method* followed by another *Pull Up Method*). In practice, this first sequence is unnecessary, as the developer can pull up the method two classes up in the inheritance tree. There is no need to look for opportunities to first pull up the method to the immediate superclass and then *immediately* look for an opportunity to pull up the method one class more in the hierarchy tree. If the developer wants to move a method to its super-super-class, there is no need to first move it to the super-class and then moving it to the super-super-class (if this later movement is possible).

This led us to the first simplification rule in the DFA representing refactoring sequences. Whenever there is a sequence of two applications of the *Pull Up Method* refactoring pattern involving the same method, the path can be simplified in a way that the method is pulled up to the higher superclass in the inheritance tree (the superclass in the second *Pull Up Method* occurrence). This rule can be expressed as follows:

$$S.pu(m,s).pu(m,s') \equiv S.pu(m,s') \tag{8.3}$$

Therefore, in the initial DFA, the *Pull Up Method* followed by Pull Up Method sequence can be removed: $pu(m,s).pu(m,s')$. Note that this simplification only excludes the search for opportunities of a sequence composed by two *Pull Up Method* refactoring patterns applied to the same method. There is still the need to look for refactoring opportunities for applying each of the *Pull Up Method* individually. Also, each application of a refactoring pattern has to be evaluated in terms of its pre-conditions. This can be *lazily* evaluated, as it is expensive to compute beforehand all the possible refactoring opportunities.

8.4.2.2 Simplification: λ – Inline Sequences

A second path that can be simplified occurs when a *Pull Up Method* is followed by an *Inline Method* refactoring pattern. The application of *Inline Method* replaces all the calls to the method by the contents of the method. Moving the method to the superclass and then inlining it in the same refactoring sequence does not make sense, and therefore this possibility is rejected. This simplification can be expressed as follows:

$$S.pu(m,s).im(m) \equiv S.im(m) \tag{8.4}$$

In fact, after applying the *Inline Method*, the application of the remaining selected refactoring patterns in Table 8.1 does not make sense. Therefore, the sequences can be

simplified to the direct application of *Inline Method*, instead of applying first the other refactoring patterns.

In summary, whenever a sequence contains a *Pull Up Method*, a *Rename Method*, a *Move Method* or a *Push Down Method* refactoring pattern followed by an *Inline Method* operating on the same method, the sequence can be simplified in a way that only the *Inline Method* refactoring pattern is applied. This rule can be expressed as follows:

$$S.\lambda.im(m) \equiv S.im(m) \tag{8.5}$$

where $\lambda \in \{pu(m,s), rm(m,n), mm(m,nc), pd(m,sc)\}$

This simplification leads to the removal of all the sequences ending with an *Inline Method*: $rm(m,n).im(m)$, $mm(m,nc).im(m)$, $pu(m,s).im(m)$ and $pd(m,sc).im(m)$.

8.4.2.3 Commutative Paths: Rename Method and Pull Up Method Sequences

Another common simplification is the occurrence of commutative paths. By observing the sequence *Pull Up Method* followed by the application of *Rename Method* in a different order, one can conclude that it does not matter if the method is renamed before or after the *Pull Up Method* application. This leads us to the next simplification rule, which can be summarized as follows:

If there is a sequence composed by a *Pull Up Method* followed by a *Rename Method*, the inverse sequence (*Rename Method* followed by *Pull Up Method*) can be removed from the DFA (or vice-versa). This rule can be expressed as:

$$S.pu(m,s).rm(m,n) \equiv S.rm(m,n).pu(m,s) \tag{8.6}$$

Figure 8.5 shows the rule expressed as a DFA.

In this rule, the choice was to arbitrarily remove the *Rename Method – Pull Up Method* sequence: $rm(m,n).pu(m,s)$. It does not matter which one of the commutative paths is chosen for removal.

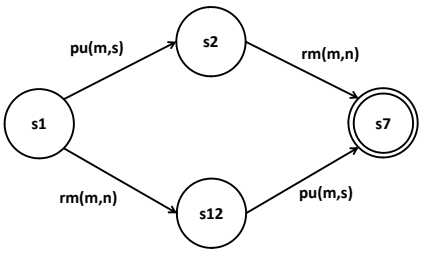

Figure 8.5: Commutative paths: *Rename Method* and *Pull Up Method* sequences

8.4.2.4 Simplification: Pull Up Method – Move Method Sequences

The next simplification refers to the case when a sequence composed of a *Pull Up Method* followed by a *Move Method* operate in the same method. The *Pull Up Method* does not change the overall result, as the method is moved again to another class. This led us to another simplification rule, stating that whenever a *Pull Up Method* is applied before a *Move Method* operating in the same method, the sequence can be simplified to the initial state followed by the application of *Move Method*. The definition is as follows:

$$S.pu(m,s).mm(m,nc) \equiv S.mm(m,nc) \tag{8.7}$$

In this case, the larger sequence $pu(m,s).mm(m,nc)$ is removed from the initial DFA.

8.4.2.5 *Inverse: Pull Up Method – Push Down Method Sequences*

One additional simplification that can be performed is the search for inverse paths, i.e. paths that reverse the effects of a previously applied refactoring. For example, if a method is pulled up from a class to a superclass and after that it is pushed down to the original class again, all the classes remain the same. Therefore, whenever a sequence of a *Pull Up Method* followed by a *Push Down Method* is applied to the same method in the same classes, the end result is the initial state:

$$S.pu(A.m, B).pd(B.m, A) \equiv S \tag{8.8}$$

This case does not occur when the classes involved in the operations are different. For example, if a method is pulled up from class A to class B and after that it is pushed down from class B to class C, the resulting state is different from the original state:

$$S.pu(A.m, B).pd(B.m, C) \not\equiv S \tag{8.9}$$

In this case, the initial *pu(m,s) . pd(m,sc)* sequence stays in the DFA, until the actual parameters are evaluated to see if it is an inverse sequence.

8.4.3 Comparing the DFAs

In summary, the *Pull Up Method* branch of the DFA was simplified with the following rules:

$$\begin{aligned}
S.pu(m, s).pu(m, s') &\equiv S.pu(m, s') \\
S.pu(m, s).im(m) &\equiv S.im(m) \\
S.pu(m, s).mm(m, nc) &\equiv S.mm(m, nc) \\
S.pu(m, s).rm(m, n) &\equiv S.rm(m, n).pu(m, s) \\
S.pu(A.m, B).pd(B.m, A) &\equiv S
\end{aligned}$$

Repeating the same approach for the *Rename Method*, *Move Method* and *Push Down Method* branches, the DFA is further simplified. The following additional rules were applied to simplify the DFA:

$$\begin{aligned}
S.rm(m, n).rm(m, n') &\equiv S.rm(m, n') \\
S.rm(m, n).mm(m, nc) &\equiv S.mm(m, nc).rm(m, n) \\
S.mm(m, nc).mm(m, nc') &\equiv S.mm(m, nc') \\
S.pd(m, sc).pu(m, s') &\equiv S \\
S.pd(m, sc).mm(m, nc) &\equiv S.mm(m, nc) \\
S.pd(m, sc).pd(m, sc') &\equiv S.pd(m, sc')
\end{aligned}$$

Figure 8.3 and Figure 8.6 show, respectively, the initial DFA and the simplified DFA. A reduction of 62% of the initial number of sequences was achieved.

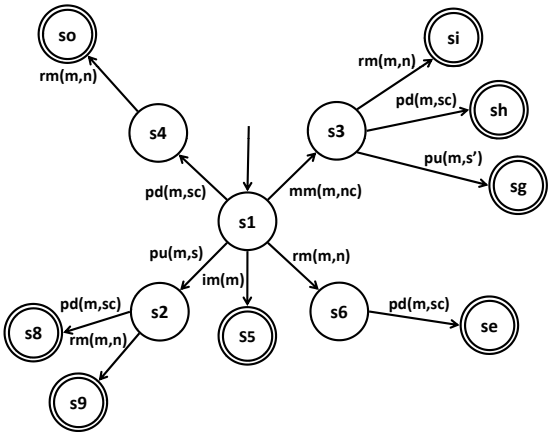

Figure 8.6: Refactoring methods simplified DFA - 8 paths

8.4.4 Sequences in the Sample Projects

The actual number of simplified sequences can vary from project to project. For a module (class aspect or interface) $\mathcal{M}$, the number of sequences can be initially computed using the nom, dit and noc metrics, as follows:

$$
\begin{aligned}
rm.rm &= nom(M) \\
rm.mm &= nom(M) \\
rm.pu &= \begin{cases} nom(M) & : \; dit(M) > 0 \\ 0 & : \; otherwise \end{cases} \\
rm.pd &= \begin{cases} nom(M) & : \; noc(M) > 0 \\ 0 & : \; otherwise \end{cases} \\
\lambda.im &= nom(M) \\
mm.\lambda &= nom(M) \\
pu.\lambda &= \begin{cases} nom(M) & : \; dit(M) > 0 \\ 0 & : \; otherwise \end{cases} \\
pd.\lambda &= \begin{cases} nom(M) & : \; noc(M) > 0 \\ 0 & : \; otherwise \end{cases} \\
im.\lambda &= 0
\end{aligned}
$$

To assess the applicability of the proposed approach, the sequences were computed for the ten sample projects presented in Chapter 9 using the number of operations in module, depth of inheritance tree and number of children metrics. Table 8.2 shows information about the reduction of the sequences, including the name of the evaluated project, the number of initial sequences, the number of simplified sequences and the percentage of reduction.

Note that this number of sequences is large because all the methods in all the classes, interfaces and aspects are being considered. In practice, however, the developer will evaluate the sequences for a small sub-set of the system modules. He can select an individual class or all the classes in a given package, for example, to get a manageable set of sequences.

Table 8.2: Reduction of Sequences on Sample Projects

Project	**Number of Initial Sequences**	**Number of Simplified Sequences**	**Reduction**
AspectJ Design Patterns	3954	1682	57%
AspectJ Examples	5475	2254	59%
AspectJ Hot Draw	54765	21906	60%
aTrack	4263	1810	58%
Jakarta Cactus	9777	3982	59%
Glassbox	3225	1316	59%
GTalkWap	1836	762	58%
Infra Red	21495	9002	58%
My SQL Connector J	40005	16284	59%
Surrogate	1401	580	59%

8.5 Tool Support

Tool support is provided for binding a grammar to refactoring patterns in a catalogue, and for creating the initial sequences for a set of refactoring patterns, and a number of levels of sequences. Figure 8.7 shows the main classes of the developed API.

Consider, for instance, the sequences generated in the example of this chapter. Five refactoring patterns are being used, which affect methods in an object-oriented language. The following code is used to create a grammar with the non-terminal *Method*, and a catalogue containing the refactoring patterns *Pull Up Method*, *Inline Method*, *Rename Method*, *Move Method*, and *Push Down Method*.

```
1 Grammar javaGrammar = new Grammar("Java Grammar");
2 javaGrammar.addNonTerminal(new NonTerminalSymbol("Method"));
3 ...
4 RefactoringCatalog fowlerCatalog = new RefactoringCatalog();
5 fowlerCatalog.add(new RefactoringPattern("Pull Up Method"));
6 fowlerCatalog.add(new RefactoringPattern("Inline Method"));
7 fowlerCatalog.add(new RefactoringPattern("Rename Method"));
8 fowlerCatalog.add(new RefactoringPattern("Move Method"));
9 fowlerCatalog.add(new RefactoringPattern("Push Down Method"));
```

The next step is to bind the catalogue with the grammar, and bind each refactoring pattern with a grammar symbol. The *CatalogGrammarBinder* is the class responsible for binding the catalogue and the grammar together. It also provides a method for binding the refactoring patterns with the grammar symbols (*bindSymbolToPatterns*), in this case, the five refactoring patterns to the *method* symbol.

```
1 CatalogGrammarBinder binder = new CatalogGrammarBinder();
2 binder.bind(fowlerCatalog, javaGrammar);
3
4 ArrayList<RefactoringPattern> methodRps = new ArrayList<
    RefactoringPattern>();
5 methodRps.add(fowlerCatalog.get("Pull Up Method"));
6 methodRps.add(fowlerCatalog.get("Inline Method"));
7 methodRps.add(fowlerCatalog.get("Rename Method"));
8 methodRps.add(fowlerCatalog.get("Move Method"));
9 methodRps.add(fowlerCatalog.get("Push Down Method"));
```

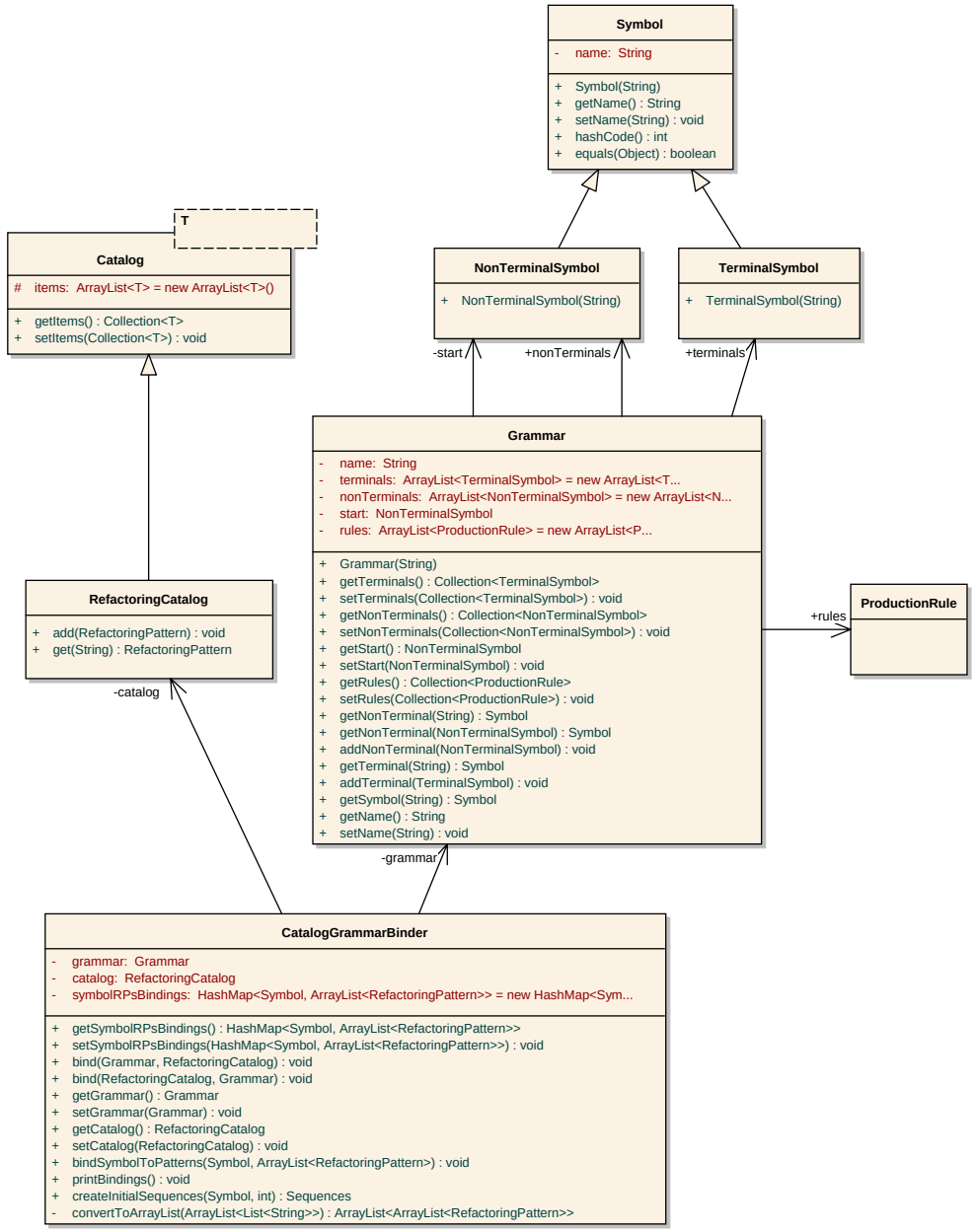

Figure 8.7: Tool support for binding grammars and catalogues of refactoring patterns, and creating the initial sequences

```
10  binder.bindSymbolToPatterns(javaGrammar.getSymbol("Method"),
        methodRps);
```

The binder creates the initial sequences, which are stored in a class named *Sequences*. The generation is performed by the *createInitialSequences* method, which receives a grammar symbol and the number of levels of successive refactoring, and generates the initial sequences. The *print* method prints the generated sequences on the standard output.

```
1  Sequences s = binder.createInitialSequences(javaGrammar.
       getSymbol("Method"), 2);
2  s.print();
```

8.6 Related Work

Mens et al. (MENS et al., 2003) describe current trends and future research regarding refactoring in general. They observe that determining where and why refactoring patterns should be applied is still an open problem. This chapter presents an approach to reduce the number of software elements to evaluate when searching for opportunities for refactoring sequences.

Tourwe and Mens (TOURWE; MENS, 2003) use logic meta-programming to search for refactoring opportunities (including sequences of refactoring) in existing software. They state that identifying opportunities for refactoring sequences requires checking opportunities for each and every possible refactoring, which could take quite too much time and should be the focus of future work. This chapter extends their work by providing mechanisms to reduce the search space, reducing the effort needed to search for opportunities for refactoring sequences.

Mens et al. (MENS; TAENTZER; RUNGE, 2005) also explore the problem of structural evolution conflicts in a formal way by using graph transformation and critical pair analysis. They show how this formalism can be used to detect and resolve refactoring conflicts. Heckel et al. (HECKEL; KUSTER; TAENTZER, 2002) establish a definition of critical pairs for typed attributed graph transformation and provide a critical pair lemma. According to them, local confluence follows from confluence of all critical pairs. Their techniques are used by the approach of this chapter to help detecting forbidden sequences and independent sequences.

Approaches for identifying refactoring opportunities (SIMON; STEINBRUCKNER; LEWERENTZ, 2001) and for evaluating the effects of refactoring on quality attributes (BOIS, 2006; BOIS; MENS, 2003) can be adapted to be used together with the approach of this chapter, in order to identify refactoring opportunities by considering refactoring sequences.

8.7 Conclusions

This chapter describes an approach for reducing the search space for refactoring opportunities, by providing mechanisms to create and simplify a DFA representing the applicable refactoring sequences in existing software.

The approach was exemplified using five refactoring patterns dealing with the manipulation of methods. The initial DFA was simplified and its size was reduced in 62% (considering the total number of paths to be evaluated) in the theoretical evaluation, and between 57-60% in the practical examples.

Additional techniques can be used to further reduce the scope of refactoring, including the careful selection of the refactoring patterns to be included in the search, the modules to be evaluated and the optimal parameters for each refactoring pattern.

Future work should focus on answering the question of which are the best sequences (according to the quality attributes) and on further techniques to reduce the search space, more specifically in the task of choosing the right parameters for the actual application of the refactoring patterns being used.

9 A CASE STUDY OF METRICS TO EVALUATE ASPECT-ORIENTED SOFTWARE QUALITY

This chapter provides metrics formal definitions and empirical data showing the value of six metrics for aspect-oriented software collected from ten open source projects. The chapter is organised as follows. Section 9.1 describes the main motivation for this chapter. Section 9.2 describes the selected metrics, the selected projects, and the computed statistics. Section 9.3 describes a formal definition of the metrics, usage scenarios of the metrics and empirical data. Section 9.4 shows data correlation between the metrics. Section 9.5 shows how the metrics can be used to spot shortcomings. Section 9.6 describes related work and finally, Section 9.7 concludes the chapter.

9.1 Introduction

Aspect-Oriented Software Development (AOSD) aims at providing abstraction and composition mechanisms to better modularise crosscutting concerns (KICZALES et al., 1997; ELRAD; FILMAN; BADER, 2001). These concerns often cannot be clearly decomposed from the rest of the software and their modularisation using object-oriented techniques usually results in either scattering or tangling of the resulting software.

The use of software metrics can help to evaluate various quality attributes of aspect-oriented software, such as modularity, reusability and size. For example, size metrics can support the identification of modularisation problems: large modules can be broken into smaller ones with fewer responsibilities or have their features merged into other modules.

Metrics adapted from the widely known and used metrics for object-oriented software (CHIDAMBER; KEMERER, 1994) have already been used in experimental studies on AOSD (CACHO et al., 2006; CASTOR FILHO; GARCIA; RUBIRA, 2005; GREENWOOD; BLAIR, 2006), where the original object-oriented metrics were extended to be paradigm-independent, generating comparable results (CASTOR FILHO et al., 2006). Up to date, these metrics have been informally described (CECCATO; TONELLA, 2004), their properties have not been analysed and typical values of these metrics for actual and practical software are not available in the literature.

This chapter complements these previous works by providing, for a sub-set of those metrics, formal definitions of metrics and empirical data collected from a set of widely available aspect-oriented (AO) projects, and a set of usage guidelines. These definitions can be used to improve the accuracy of quantitative assessment of aspect-oriented software by reducing the ambiguity normally present in informal descriptions. The usage scenarios can show the relation of metrics with quality attributes.

Two sets of metrics are considered:

1. Metrics adapted from Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) by Zakaria and Hosny (ZAKARIA; HOSNY, 2003), Santanna et al. (SANTANNA et al., 2003), and Ceccato and Tonella (CECCATO; TONELLA, 2004): lines of code (*locc*), number of operations in module (*nom*), depth of inheritance tree (*dit*) and number of children (*noc*);

2. Metrics specifically defined for aspect-oriented software: crosscutting degree of an aspect (*cda*) and coupling on advice execution (*cae*) (CECCATO; TONELLA, 2004).

9.2 Selected Metrics, Projects, and Statistics

This section discusses the metrics selected for this study, the projects used to collect empirical data, and the computed statistics.

9.2.1 Selected Metrics

Many aspect-oriented metrics can be formally defined, evaluated and their values for open source projects interpreted. However, to take a broad perspective of aspect-oriented metrics and for the sake of brevity, this chapter focus only on six of these metrics:

- Lines of Code (*locc*)
- Number of Operations in Module (*nom*)
- Depth of Inheritance Tree (*dit*)
- Number of Children (*noc*)
- Crosscutting Degree of an Aspect (*cda*)
- Coupling on Advice Execution (*cae*)

The first four selected metrics (*locc*, *nom*, *dit* and *noc*) are used to measure size and use of inheritance and can be the basis for more complex metrics. The other two metrics (*cda* and *cae*) show how many modules an aspect affects and also how many aspects affect each module. These two coupling metrics provide basic information about the influence of aspects in the overall design.

Other metrics, such as cdc, cdo and cdl deal with the diffusion of a given concern over components, operations and lines of code. The problem with these metrics is that the definition of a concern is somewhat fuzzy and the automation of the metric is not feasible. Therefore, they are not discussed in this chapter.

Another metric used in this chapter is the ratio between the mean value of the metric μ for aspects and the mean value of the metric μ for classes. Values of this metric higher than one indicates that the mean value of the μ metric is higher in the aspects than in the classes. On the opposite way, values below one denote that the metric values are higher in the classes. A ratio value of one indicates that the mean values for this metric are equal in aspects and classes. This ratio is used to compare the values of the metric in aspects and in classes (which projects have a higher value of a chosen metric for aspects than for classes or which ones have lower values for aspects).

The ratio between the mean value of the metric μ for aspects and the mean value of the metric μ for classes can be defined as: Let $\overline{x}(\mu(aspects))$ be the mean value for a

metric μ for all the aspects and $\overline{x}(\mu(classes))$ be the mean value for a metric μ for all the classes, the ratio between the mean value of the metric μ for aspects and the mean value of the metric μ for classes (denoted by $ratioMean$) is: $\overline{x}(\mu(aspects))/\overline{x}(\mu(classes))$.

9.2.2 Selected Projects and Computed Statistics

This section briefly describes the projects used to provide empirical data used as examples in Section 9.3. Ten projects were selected from open source repositories, considering the number of users, and different domains, aiming at selecting projects that are reasonably stable and that have a significant user base. The use of heterogeneous projects intends that the collected values for the metrics represent typical values for aspect-oriented software.

Table 9.1 shows summary information (name, description, version, size, and URL) of the selected projects.

Table 9.1: Summary of selected projects

Name	**Desc.**	**Version (locc)**
1. AspectJ Design Patterns	Implementation of the GoF Patterns.	v1.1 (2,344)
URL: `http://www.cs.ubc.ca/~jan/AODPs/`		
2. AspectJ Examples	Examples of the AspectJ distribution.	AJ5 (2,878)
URL: `http://www.eclipse.org/aspectj/`		
3. AspectJ Hot Draw	An aspect-oriented version of the JHotDraw graphics framework.	v0.3 (23,051)
URL: `http://sourceforge.net/projects/ajhotdraw/`		
4. aTrack	Bug Tracking Application.	CVSHead (2,221)
URL: `https://atrack.dev.java.net/`		
5. Jakarta Cactus	Test framework for server-side java code.	v1.3 (5,244)
URL: `http://jakarta.apache.org/cactus/`		
6. Glassbox	Troubleshooting agent for Java applications.	v1.0a2 (1,562)
URL: `http://www.glassbox.com/`		
7. GTalkWap	GoogleTalk access from WAP-enabled devices.	v1.0b (1,013)
URL: `http://sourceforge.net/projects/gtalkwap`		
8. Infra Red	Performance Monitoring Tool for Java/J2EE.	v2.3 (13,888)
URL: `http://sourceforge.net/projects/infrared`		
9. My SQL Connector J	MySQL Native Java driver.	v5.0 (40,755)
URL: `http://www.mysql.com/products/connector/j/`		
10. Surrogate	Unit testing framework.	v1.0RC1 (806)
URL: `http://sourceforge.net/projects/surrogate`		

The *aopmetrics* tool[1] was used to collect the metric values for the selected projects. For each metric, the *mean* was selected as a measure of central tendency and the *standard*

[1] Available at `http://aopmetrics.tigris.org`

deviation as a measure of dispersion. The values were grouped by project and by module type (aspect or class). For each metric, histograms were created for the values for aspects and for classes. As the sample data is different for each histogram, the Shimazaki's method (SHIMAZAKI, 2006) was used to select the bin size (the bin size represents the size of each category in the histogram).

The accuracy of samples is usually measured using margin of error (SNEDECOR et al., 1989). The amount by which the values obtained from the sample will differ from the true population values rarely exceeds one divided by the square root of the size of the sample ($1/\sqrt{n}$), where n represents the number of elements in the sample. In this chapter, the margin of error for the analysis related to values for aspects is 8% and for classes it is 3%.

9.3 Formal Definitions of Metrics and Empirical Data

This section provides, for each of the six selected metrics, their formal definition, usage scenarios and empirical data: lines of code (*locc*), number of operations in module (*nom*), depth of inheritance tree (*dit*), number of children (*noc*), crosscutting degree of an aspect (*cda*) and coupling on advice execution (*cae*). Table 9.2 summarizes the number of aspects and classes per selected project, presenting an overall feeling of the size of the selected projects (in terms of aspects and classes).

Table 9.2: Modules in the project

Project Name	**#Classes**	**#Aspects**	**% of Aspects**
AspectJ Design Patterns	104	40	27.8%
AspectJ Examples	56	27	32.5%
AspectJ Hot Draw	357	10	2.7%
aTrack	53	28	34.6%
Jakarta Cactus	93	1	1.1%
Glassbox	28	24	46.2%
GTalkWap	25	2	7.4%
Infra Red	158	11	6.5%
My SQL Connector J	149	1	0.7%
Surrogate	19	3	13.6%
Total	**1092**	**147**	**11.9%**

Each metric is described using the following structure:

- *Informal Definition:* In the introduction of each metric, an informal introduction is provided to describe the meaning of the metric;
- *Formal Definition:* Set theory is used to describe the metric in a formal way;
- *Usage:* The usage scenarios for the metric are discussed, together with scenarios for the combination with other metrics;
- *Empirical Data:* A set of summary statistics for the values of the metric in the selected projects is provided. Also, a brief discussion of the metric values in the selected projects is conducted;

9.3.1 Lines of Code

This metric counts the number of *lines of code* (*locc*). The Java and AspectJ grammars[2] were used to define the components needed to compute this metric. The following considerations from Stochmialek (STOCHMIALEK, 2009) are used to compute the *locc* of a module:

- comments, javadocs and empty newlines are not counted;
- class, aspect, method and advice headers are counted as a line;
- a new line is created for a closing curly braces;
- string constants are counted as a single line.

In AspectJ, aspects can be composed of several elements, including those that can be also elements of classes. Aspects can contain declare constructions, advices, inter-type method/constructor declarations, inter-type field declarations, inner classes/aspects/interfaces, enumerations, constructors, fields and methods.

9.3.1.1 Formal Definition

Let $\mathcal{O}$ be the set of declare constructions, inter-type field declarations, enumerations and fields of a module, $\mathcal{I}$ be the set of inner classes/aspects/interfaces of a module and $\mathcal{M}$ be the set of advices, methods, constructors and inter-type method/constructor declarations of a module. Consider that the set $\mathcal{A}$ is composed of several elements $(a_1, ..., a_i)$ where $i = |\mathcal{A}|$ and the $\mathcal{IC}$ set is composed of several elements $(ic_1, ..., ic_j)$ where $j = |\mathcal{IC}|$. The $locc(m) : Module \rightarrow \mathbb{N}$ of a module *m* can be computed by:

$$locc(m) \quad = \quad 1 + |\mathcal{O}| + \sum_{i=0}^{n} locc(a_i) + \sum_{j=0}^{m} locc(ic_j) \tag{9.1}$$

In this case, the class or aspect declaration is counted as one line of code. Each declare construction, inter-type field declaration, enumeration and field is counted as one line of code (their sum is denoted by the cardinality of the $\mathcal{O}$ set). Advices, inter-type method/constructor declarations, constructor and methods are composed of statements (conditionals, loops[3]). Considering each statement in a single line, the *locc* of such constructions can then be computed by the number of carriage returns in the body of each construction (tools for collecting metrics usually starts by formatting the source code).

9.3.1.2 Usage

Metrics that count lines of code are usually used as indicators of effort, productivity and cost (FENTON; PFLEEGER, 1997). In the measurement of effort, the *locc* metric is used to allow comparisons between different projects or systems. This metric is also used in productivity measurements (such as $locc/hour$) or costs ($cost/locc$), for example.

The absolute value of *locc* of a class or aspect can be used as a rough indicative of how much effort was put into developing and maintaining an aspect, and as an indicative of complexity (FENTON; PFLEEGER, 1997).

The following considerations can be made to relative measures using *locc*:

[2]Java Grammar at `https://javacc.dev.java.net/`, AspectJ Grammar at `http://abc.comlab.ox.ac.uk/documents/scanparse.pdf`

[3]See the full Java grammar at `https://javacc.dev.java.net/` for more details of all possible statements.

- The *locc* metric can be used to express rates regarding quality attributes, such as defects per *locc* (BRIAND; MORASCA; BASILI, 1996).

- The *locc* is also used to measure the size of methods (*locc* per method) and the density of documentation of a module (lines of comments per *locc*) (FENTON; PFLEEGER, 1997).

- If flexibility is an important requirement of the system being developed (when developing a framework, for example), the overall value of *locc* might be higher than if a system with few extension points is developed. The developer can consider a higher value for the threshold of the *Large Aspect* shortcoming, for example.

Considering the combination of the *locc* metric with the other metrics discussed in this chapter, the following usage guidelines are described:

- The relation of $locc/nom$ can be used as an indicative of the size of operations. High *locc* values with low values of *nom* (number of operations in a given module, as described in the next metric entry), can be an occurrence of the *Long Method* shortcoming (FOWLER et al., 1999), because this shortcoming is described in terms of the size of a method. Modules with high *locc* values and high *nom* values can be an occurrence of the *Large Aspect* or of the *Large Class* shortcomings (FOWLER et al., 1999), because both shortcomings are described in terms of the size of aspects and classes.

- Aspects with high *locc* values and with low *cda* (crosscutting degree of an aspect) values can denote that the aspect has state or behaviour that is not dealing with crosscutting concerns, because the aspect has a large size, but do not affect other modules. Such cases can be occurrences of the *Lazy Aspect* shortcoming. Part of the state and behaviour of this aspect can be moved to a new or to an existing class, thus reducing the size of the aspects to the minimum needed to implement the crosscutting behaviour (as aspects are abstractions created to deal with crosscutting concerns). The aspect can use association mechanisms to access these features.

- Another possible combination is in terms of *cae* values. Aspects or classes with low *locc* values and high *cae* values can be occurrences of the *Aspect Interaction* shortcoming. The smaller the values of *locc* and the higher the values of *cae*, the higher is the probability of aspects interactions.

9.3.1.3 Empirical Data

Table 9.3 summarizes the statistical data collected from the ten selected projects. The first two columns show the mean and standard deviation for aspects, and the next two columns show the same information for classes and the last one shows the ratio between the mean value of *locc* for aspects and the mean value of *locc* for classes. Figure 9.1 shows the values of *locc* for aspects and for classes[4]. The x-axis shows the value of $locc$ and the y-axis shows the relative frequency of each category in the projects.

In the selected projects, the core functionality is defined in classes. The aspects modularise concerns that would be otherwise scattered over the classes. The computed mean value of *locc* for classes in the selected projects is 94.3 and the standard deviation is

[4]In the classes histogram, 95% of the available data is shown to more easily compare the values for aspects and classes (i.e. the x-range is smaller than the maximum value of *locc* for classes).

Table 9.3: Summary statistics for *locc* values

Project Name	$\overline{locc}$ (Aspects)	σ	$\overline{locc}$ (Classes)	σ	$\overline{locc}$ rat.
AspectJ Design Patterns	19.7	14.1	13.9	9.5	1.42
AspectJ Examples	33.7	35.9	34.7	39.3	0.97
AspectJ Hot Draw	18.0	10.4	62.8	88.7	0.29
aTrack	33.5	46.4	22.1	18.8	1.51
Jakarta Cactus	88.0	0.0	54.3	64.1	1.62
Glassbox	40.5	35.7	18.9	16.6	2.14
GTalkWap	11.0	7.1	38.2	33.6	0.29
Infra Red	33.7	38.2	84.5	97.4	0.40
My SQL Connector J	186.0	0.0	271.9	668.4	0.67
Surrogate	7.0	6.9	41.0	54.3	0.17
Total	30.4	35.5	94.3	291.8	0.32

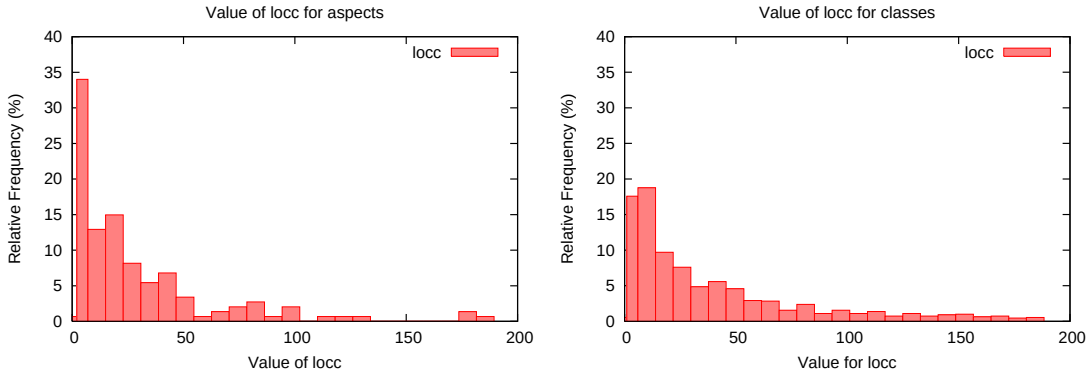

Figure 9.1: Value of *locc* for aspects and classes

291.8. The mean value of *locc* for the aspects is 30.5 and the standard deviation is 35.5. The mean values of *locc* in the selected projects are, in general, higher for classes; also, the variability for classes is higher than for aspects.

If the ratio between the mean *locc* of aspects and the mean *locc* of classes is analysed, one can see that there are projects with a ratio lower than one (classes are bigger, in terms of *locc*) and projects with ratio higher for aspects. Projects presenting a low ratio include Surrogate (0.17), AspectJ Hot Draw (0.29), GTalkWap (0.29), Infra Red (0.40), My SQL Connector J (0.67) and AspectJ Examples (0.97). In these projects, the core functionality of the application is modularised in classes and the aspects are used to encapsulate auxiliary concerns (such as logging, tracing and policy enforcement), application of design patterns or other infra-structure aspects.

There are projects in which the ratio is bigger than one (i.e. the aspects are bigger than the classes). This usually occurs when the application is heavily based on aspects, such as the aTrack (1.51) and the AspectJ Design Patterns (1.42) projects or when the application is designed to be plugged into another using load-time weaving. In this sense, the *locc* of the affected classes is not being computed. This case occurs in the Glassbox (2.14) project. One last project with a high ratio is the Jakarta Cactus (ratio of 1.62), which has 93 classes and only one aspect. This single aspect is responsible for logging every entry and exit of methods and is quite long. It could be simplified by reducing the duplication

inside its advices.

Considering the collected data, 95% of the modules have a *locc* smaller than 250 and 85% are smaller than 100 lines of code. Developers tend to define small classes to improve the understandability of the modules and reduce the defects per module (BRIAND; MORASCA; BASILI, 1996). There are however, classes with high values for *locc*. Querying the metric values of the selected projects, the maximum value of *locc* is 4374. Considering that that the mean *locc* is 30 and the maximum value for the *locc* in the aspects is 186, 4374 lines of code is high number for the size of a class.

9.3.2 Number of Operations in Module

The *number of operations in module* (*nom*) metric counts the number of operations in a given module (CECCATO; TONELLA, 2004). The *nom* of classes is defined as the number of methods of a given class (CHIDAMBER; KEMERER, 1994). When dealing with aspects, besides methods there is the need to also consider advices and inter-type declarations. So, in this chapter, a slightly different formula for the *nom* is defined, specifically for aspects. The *nom* metric relates directly to the complexity of modules, since advices are method-like constructs that provide a way to express crosscutting actions at the join points that are captured by a pointcut (BERG; CONEJERO; CHITCHYAN, 2005). Informally, the value of *nom* for a module is given by sum of the number of its methods, advices, inter-type method declarations and inter-type constructor declarations.

9.3.2.1 Formal Definition

Let $\mathcal{M}$ be the set of methods, $\mathcal{A}$ the set of advices, $\mathcal{MD}$ the set of inter-type method declarations and $\mathcal{CD}$ the set of inter-type constructor declarations of an *m* module. The *nom* of the module *m* is given by a function $nom(m) : Module \rightarrow I\!N$:

$$nom(m) = |\mathcal{M}| + |\mathcal{A}| + |\mathcal{MD}| + |\mathcal{CD}| \tag{9.2}$$

9.3.2.2 Usage

The following viewpoints were adapted from Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) and are applicable to the *nom* metric:

- The number and the complexity of advices in a class indicate how much time and effort is needed to develop and maintain the aspect;
- Aspects with large numbers of advices are likely to be more application specific, limiting the possibility of reuse.
- One of the original viewpoints is that the larger the number of methods in a class, the greater the potential impact on children, as children will inherit all the methods defined in a class. The impact of this viewpoint is not as high for aspects (as in AspectJ, for instance, the sub-aspects do not redefine the advices of super-aspects), as the advices of super-aspects do not influence much the complexity of sub-aspects.

Other considerations regarding the *nom* metric are:

- Classes and aspects with low values of *nom* can be inspected to see if they are occurrences of the *Lazy Aspect* (MONTEIRO; FERNANDES, 2005a; PIVETA et al., 2005) or the *Lazy Class* (FOWLER et al., 1999) shortcoming. These cases occur

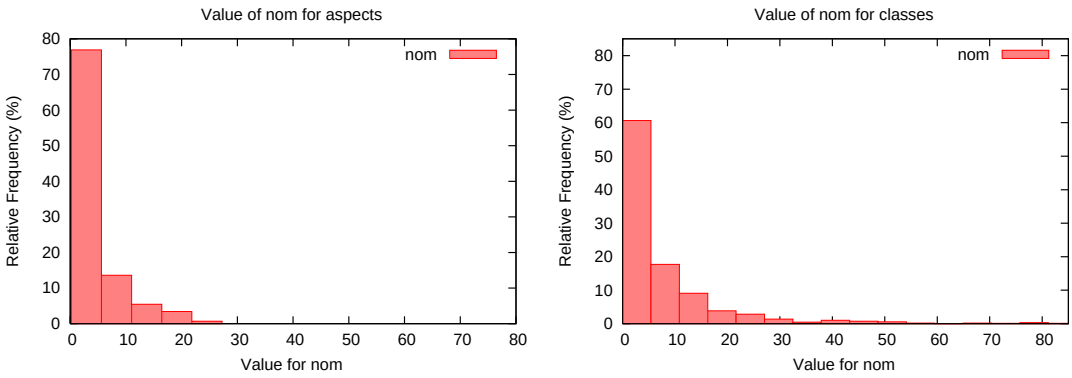

Figure 9.2: Value of *nom* for aspects and classes

if an aspect or class has few responsibilities, and its elimination can be beneficial. Sometimes, this responsibility reduction is related to previous refactoring or to unexpected changes in requirements (planned changes that did not occur, for instance).

- Classes and aspects with high values of *nom* can be occurrences of the *Large Classes* (FOWLER et al., 1999) or *Large Aspects* (PIVETA et al., 2005) shortcomings. When an aspect encapsulates more than one concern, it should be divided in as many aspects as there are concerns. This shortcoming is usually discovered when the developer finds several unrelated members (fields, pointcuts, inter-type declarations) in the same aspect (PIVETA et al., 2006a).

- In terms of reusability, modules with high values of *nom* can suffer from the *Refused Bequest* shortcoming (FOWLER et al., 1999)[5], in which the sub-classes inherit methods that are not used.

- In terms of modularity, the higher the value of *nom*, the higher is the likelihood that those methods are responsible to deal with different concerns. Further research is needed to correlate *nom* with cohesion metrics.

When considering the combinations of this metric with the other metrics discussed in this chapter, the following usages can be made:

- The cda/nom metric can be an indicative of how much influence (in terms of affected modules) an aspect has. High values of cda/nom indicate that the advices affect several modules.

- The cae/nom metric can be used to see how much the aspects influence the overall behaviour of a given module. The value of cae/nom is directly proportional to the influence of aspects in a module.

9.3.2.3 Empirical Data

Table 9.4 shows summary statistics related to the *nom* metric. Figure 9.2 shows histograms for the values of *nom* for aspects and classes. The x-axis shows the value of nom and the y-axis shows the relative frequency of each category in the projects.

[5] A more precise name for this shortcoming would be *Ignored Bequest*

Table 9.4: Summary statistics for *nom* values

Project Name	$\overline{nom}$ (Aspects)	σ	$\overline{nom}$ (Classes)	σ	$\overline{nom}$ rat.
AspectJ Design Patterns	2.8	2.7	2.1	1.2	1.34
AspectJ Examples	5.3	5.3	4.9	4.6	1.08
AspectJ Hot Draw	2.7	2.5	9.2	12.1	0.29
aTrack	4.1	6.1	3.8	4.2	1.08
Jakarta Cactus	5.0	0.0	7.2	9.2	0.69
Glassbox	5.3	6.0	2.8	2.7	1.91
GTalkWap	1.5	0.7	5.5	4.4	0.27
Infra Red	2.4	2.0	10.7	15.1	0.22
My SQL Connector J	17.0	0.0	19.4	47.8	0.96
Surrogate	0.3	0.6	5.8	7.0	0.05
Total	3.9	4.8	9.3	21.0	0.42

The values for the *nom* metric are highly correlated to those of *locc* (Section 9.4 describes this in more details). The mean value of *nom* for classes is 9.3 and for aspects is 3.9, denoting that the number of operations in classes is larger than in the aspects in the selected sample. Also, the variability of this metric in the classes is higher than in the aspects. In the selected projects, the computed standard deviation of classes is 21 and in aspects it is 4.8. Also, the maximum value of *nom* for classes is 313 and for aspects it is 23.

The majority of modules (81.3%) have a maximum of ten operations (methods, advices, inter-type method declarations or inter-type constructor declarations), while 11% have between 11 and 20 operations and 7.6% have more than 20 operations. Considering only classes, there are 78.3% of them with less than 10 methods, 12.4% with 11 to 20 methods and 9.3% with more than 20 methods. The *nom* of aspects is usually smaller: 90.5% of the aspects have up to 10 operations, 8.8% have between 11 and 20 and only 0.7% (one aspect) has more than 20 operations.

The ratio between the mean of *nom* in classes and the mean of *nom* in aspects provides an indication of the proportion between the number of operations in classes and aspects. The ratio per project is below one for the Surrogate (0.05), Infra Red (0.22), GTalkWap (0.27), AspectJ Hot Draw (0.29), Jakarta Cactus (0.69) and My SQL Connector J (0.96). The last project presents a high ratio value because there is only one aspect with 17 operations. Ratio values higher than one are found in the aTrack (1.08), AspectJ Examples (1.08), AspectJ Design Patterns (1.34) and Glassbox (1.91) projects. Note that these ratio values are smaller than the ones presented for the *locc* metric.

9.3.3 Crosscutting Degree of an Aspect

The *crosscutting degree of an aspect* (*cda*) metric counts the number of modules affected by advices, declare constructions, declared annotations, inter-type method declarations and inter-type constructor declarations in a given aspect (CECCATO; TONELLA, 2004).

9.3.3.1 Formal Definition

Let $\mathcal{AA}$ be the set of modules affected by advices of an aspect α, $\mathcal{AD}$ the set of modules affected by declare constructions of α, $\mathcal{AN}$ the set of modules annotated by the aspect α, $\mathcal{AI}$ the set of modules affected by the inter-type declarations of α. A function $cda(\alpha) : Aspect \rightarrow I\!N$ that computes the crosscutting degree of an aspect can be defined as the cardinality of the union of the $\mathcal{AA}$, $\mathcal{AD}$, $\mathcal{AN}$ and $\mathcal{AI}$ sets:

$$cda(\alpha) = |\mathcal{AA} \cup \mathcal{AD} \cup \mathcal{AN} \cup \mathcal{AI}| \tag{9.3}$$

9.3.3.2 Usage

The crosscutting degree of an aspect metric can be used as an indicator of separation of concerns (GARCIA et al., 2006). The following usages can be considered:

- High values of *cda* are desirable (CECCATO; TONELLA, 2004), as the *cda* metric indicates how many modules an aspect affects and how useful the aspect is.
- Ceccato and Tonella (CECCATO; TONELLA, 2004) point out that while high values of *cda* are desirable, the number of explicitly named modules in the pointcut expression of an aspect must be kept low (CECCATO; TONELLA, 2004). This case is expressed as a guideline named *Use semantic based pointcuts*, which suggests the use of annotations or references to implemented interfaces to express the pointcut expressions (Appendix A).
- If the *cda* value is equal to one, it means that the aspect affects only one class or aspect. Aspects are usually used to implement a concern that would be scattered over several classes. If there is no scattering, it is better to use classes instead. The developer can evaluate if it is better to inline the aspect or use inheritance or association mechanisms to separate the concerns encapsulated by the aspect.
- Section 9.3.1 discusses combinations with the *locc* metric and Section 9.3.2 combinations with the *nom* metric.

9.3.3.3 Empirical Data

Table 9.5 shows summary statistics of the *cda* metric. Figure 9.3 shows the values of *cda* in the selected projects. Note that this metric only applies to aspects, not to classes. The x-axis shows the value of cda and the y-axis shows the relative frequency of each category in the projects.

The values of *cda* are low in general (72% of the aspects have a *cda* value of three or less), but the values can be high and can vary according to the nature of the concerns being encapsulated by the aspects (with a maximum of 78 in the selected projects). Logging and tracing aspects are more likely to have high values of *cda* than other aspects. Higher values of *cda* indicate that the aspect is a valuable entity. This happens because if the concern is being implemented as a class, calls to its methods have to be scattered over other classes.

9.3.4 Coupling on Advice Execution

The *coupling on advice execution* (*cae*) metric counts the number of aspects containing advices that are possibly triggered by the execution of operations in a given module - i.e. the number of aspects affecting the module (CECCATO; TONELLA, 2004).

Table 9.5: Summary statistics for *cda* values

Project Name	$\overline{x}$	σ	**Min**	**Max**
AspectJ Design Patterns	2.8	5.9	0	38
AspectJ Examples	3.5	4.8	0	20
AspectJ Hot Draw	3.6	5.4	1	18
aTrack	13.3	21.2	0	75
Jakarta Cactus	68.0	0.0	68	68
Glassbox	5.17	9.5	0	33
GTalkWap	3.0	1.4	2	4
Infra Red	2.0	4.7	0	15
My SQL Connector J	78.0	0.0	78	78
Surrogate	1.3	2.3	0	4
Total	6.2	13.9	0	78

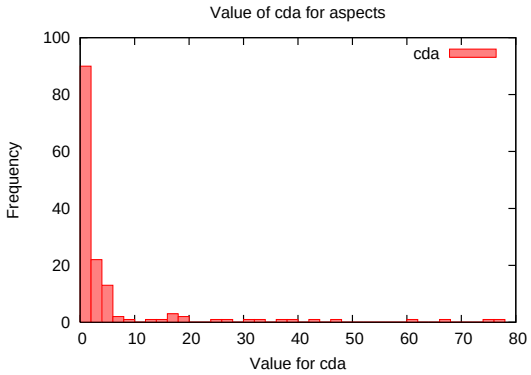

Figure 9.3: Value of *cda* for aspects

9.3.4.1 Formal Definition

Let $\mathcal{EA}$ is the set of aspects that advises a module m, $\mathcal{ED}$ the set of aspects that add declare (parents or implements) constructions to m, $\mathcal{EN}$ the set of aspects that add annotations to m, $\mathcal{EI}$ the set of aspects that define inter-type declarations to m and $\mathcal{EE}$ the set of aspects that declare error or warnings to m. The $cae(\mathrm{m}) : Module \rightarrow I\!N$ function can be defined as the cardinality of the union of the $\mathcal{CE}$ elements:

$$cae(m) = |\mathcal{EA} \cup \mathcal{ED} \cup \mathcal{EN} \cup \mathcal{EI} \cup \mathcal{EE}| \tag{9.4}$$

9.3.4.2 Usage

The values of the *cae* metric can be used to check if there is an occurrence of the *Aspect Interaction* shortcoming. The following considerations can be made:

- Low values of *cae* are good, as the higher the *cae* value, the more coupled is the class to the aspects that affect it (CECCATO; TONELLA, 2004). If a module has a *cae* with a zero value, it means that the module is not affected by aspects.

- Classes with a *cae* value of two or more can have interactions between the aspects (DOUENCE; FRADET; SUDHOLT, 2002), which can possibly lead to precedence conflicts or incompatibilities between the applied aspects. The developer should be aware of such interactions in order to evaluate if they are being handled correctly.

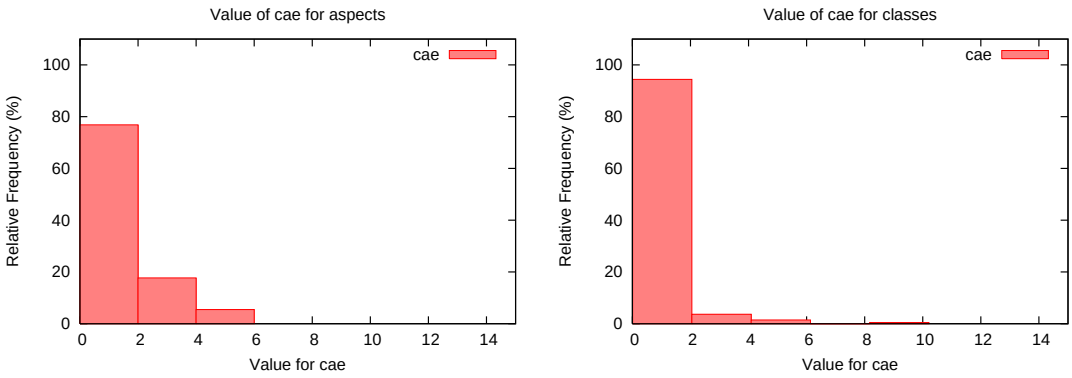

Figure 9.4: Value of *cae* for aspects and classes

- Combinations with the *locc* metric are discussed in Section 9.3.1 and combinations with the *nom* metric in Section 9.3.2.

9.3.4.3 Empirical Data

Table 9.6 shows summary statistics for the values of *cae* in aspects and classes. Figure 9.4 shows histograms for the *cae* values for aspects and classes. The x-axis shows the value of cae and the y-axis shows the relative frequency of each category in the projects.

In the selected projects, the aspects affect classes more than affect other aspects. The only exception is the Glassbox project, which uses load time weaving to attach its aspects to a target Java application, so the values for the *cae* metric in the affected classes can only be computed at load time.

Table 9.6: Summary statistics for *cae* values

	$\overline{cae}$	σ	$\overline{cae}$	σ	$\overline{cae}$ rat.
Project Name	(Aspects)		(Classes)		
AspectJ Design Patterns	0.1	0.4	0.5	0.6	0.2
AspectJ Examples	0.5	0.6	1.4	1.5	0.3
AspectJ Hot Draw	0.0	0.0	0.1	0.3	0.0
aTrack	3.3	1.5	4.4	1.9	0.7
Jakarta Cactus	0.0	0.0	0.7	0.4	0.0
Glassbox	3.0	1.4	1.5	0.9	2.0
GTalkWap	0.0	0.0	0.2	0.6	0.0
Infra Red	0.0	0.0	0.1	0.3	0.0
My SQL Connector J	0.0	0.0	0.5	0.5	0.0
Surrogate	0.0	0.0	0.2	0.4	0.0
Total	1.2	1.7	0.6	1.2	2.2

Considering the ratio between the mean value of *cae* for aspects and the mean value of *cae* for classes, all the projects - except the Glassbox - have a ratio lower than one. Six projects have a zero ratio (GTalkWap, AspectJ Hot Draw, Jakarta Cactus, Surrogate, Infra Red and My SQL Connector J), three projects have a ratio lower than one (classes are more affected than aspects), including the AspectJ Design Patterns (0.23), AspectJ Examples (0.34) and aTrack (0.74) whereas the Glassbox has a ratio of 1.97.

Note that the values for this metric are quite low, which indicates that interactions among aspects is not very common. Considering all the selected projects but Glassbox

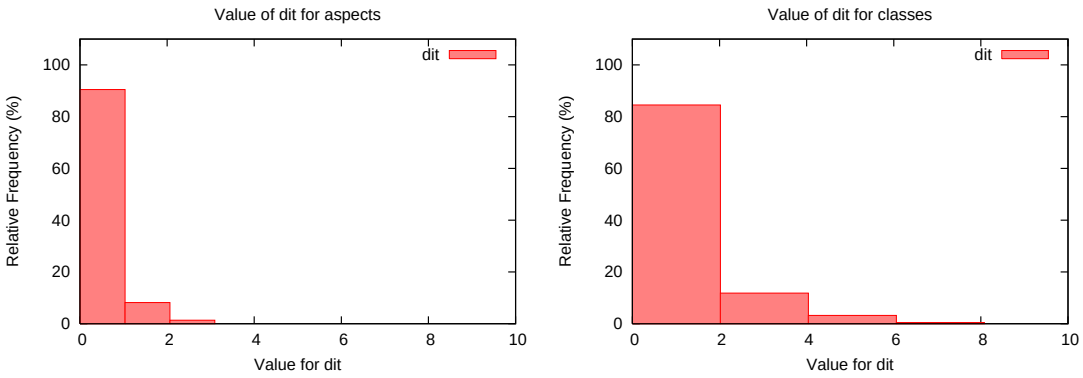

Figure 9.5: Value of *dit* for aspects and classes

and aTrack, only 2.1% of the aspects have a *cae* value higher than one. However, for the Glassbox and aTrack projects, interactions can introduce precedence issues as 91% of the aspects in the Glassbox project and 81% of the aspects in the aTrack project have $cae > 1$. The mean value of the *cda* metric for the selected projects is higher that the values of *cae*, indicating that the aspects are being used to modularise concerns that the corresponding object-oriented alternative would be otherwise scattered among several classes.

9.3.5 Depth of Inheritance Tree

The value for *depth of inheritance tree* (*dit*) is given by the longest path from a module to the class/aspect hierarchy root (CHIDAMBER; KEMERER, 1994; CECCATO; TONELLA, 2004). It is computed by counting the number of inheritance levels, from the module to the root class/aspect.

9.3.5.1 Formal Definition

Considering a function $s(x) : Module \rightarrow Module$ that computes the super-class or super-aspect of a given module, the value of *dit* is given by:

$$dit(m) \quad = \quad \begin{cases} dit(s(m)) + 1 & : \quad m \neq rootClass \\ 0 & : \quad otherwise \end{cases}$$

9.3.5.2 Usage

The following viewpoints are adapted from Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) and from Ceccato and Tonella (CECCATO; TONELLA, 2004):

- The higher the *dit* of an aspect, the more inherited methods it has and usually the more complex the aspect is, as the developer may have to understand not only the aspect but also the super-aspects or super-classes.

- Aspects with high values for *dit* are commonly project specific, whilst abstract aspects are usually more reusable across different projects.

9.3.5.3 Empirical Data

Table 9.7 shows the values for *dit* in the selected projects. Figure 9.5 shows histograms for aspects and classes. The x-axis shows the value of dit and the y-axis shows the relative frequency of each category in the projects.

Table 9.7: Summary statistics for *dit* values

Project Name	$\overline{dit}$ (Aspects)	σ	$\overline{dit}$ (Classes)	σ	$\overline{dit}$ rat.
AspectJ Design Patterns	0.4	0.5	0.5	1.4	0.8
AspectJ Examples	0.2	0.4	0.7	0.9	0.3
AspectJ Hot Draw	0.0	0.0	1.5	1.6	0.1
aTrack	0.4	0.7	1.1	1.7	0.4
Jakarta Cactus	0.0	0.0	0.9	1.1	0.0
Glassbox	1.1	1.1	1.4	1.1	0.8
GTalkWap	0.0	0.0	0.7	0.7	0.0
Infra Red	0.4	0.5	0.4	0.7	1.0
My SQL Connector J	0.0	0.0	1.0	1.3	0.0
Surrogate	0.7	0.6	0.8	1.0	0.9
Total	0.4	0.7	1.1	1.4	0.36

The value of the mean of *dit* for classes is 1.1 whilst for aspects the equivalent value is 0.4. Also, the dispersion in the values of this metric is higher for classes (1.4) than for aspects (0.7). The maximum value for *dit* of classes is eight and for aspects is three. The inheritance trees in the observed projects are deeper in classes than in aspects. This is expected to be true, as aspects in AspectJ can only extend classes or abstract aspects. As there are no benefits of inheriting advices, the reuse using inheritance is mainly due to the definition of abstract aspects with abstract pointcuts and advices. The sub-aspects override the abstract pointcuts to provide the concrete join points that the aspect will affect.

The ratio between the mean *dit* of aspects per mean *dit* of classes varies from zero to 0.8. In all projects the inheritance tree is deeper for classes than for aspects. In the selected projects, the value of *dit* of classes is less than two in 85.5% of the cases, within two and four in 11.8% of the classes and higher than four in 3.7% of the classes. In the case of *dit* for aspects, the maximum value of *dit* is three. In fact, only two aspects are three levels down in the inheritance tree. In the other cases, 90.5% of the aspects are root aspects or inherit from one abstract aspect or class and 8.2% have a *dit* equals to two.

Classes and aspects with high *dit* values can be inspected to search for misuse of inheritance or instances of *Refuse Bequest*. In the selected projects, only 3.7% of the classes have a *dit* value higher than four. Inspecting the classes with a *dit* higher than four, the developer can see that 75% of the classes are in the AspectJ Hot Draw project, where inheritance is heavily used. In this case, the project can be analysed to see if too much emphasis is given to inheritance instead of using associations, for example.

9.3.6 Number of Children

The *number of children* (*noc*) represents the number of direct sub-classes or sub-aspects for a given module (CHIDAMBER; KEMERER, 1994; CECCATO; TONELLA, 2004).

9.3.6.1 Formal Definition

Consider a function $s(x) : Module \rightarrow Module$ that computes the super-class or super-aspect of a given module and a set $\mathcal{M}$ representing the set of all modules of a given project. To compute the value of the number of children for a module *m*, let $\mathcal{S}$ be the set

of all modules that satisfy the predicate $\forall y \in \mathcal{M}, s(y) = m$. Thus, the metric value is given by the cardinality of the $\mathcal{S}$ set:

$$noc(m) = |\mathcal{S}| \tag{9.5}$$

9.3.6.2 Usage

The following viewpoints (CHIDAMBER; KEMERER, 1994) for the *noc* metric were adapted to the context of aspect-oriented software:

- The higher the values of *noc*, the higher are the possibilities that the aspect has been reused, since inheritance is a reuse mechanism. However, the higher is the likelihood of a *Refused Bequest* (FOWLER et al., 1999), in which the aspect does not use part of the attributes and methods defined in the super-class or super-aspect.

- Aspects with high values of *noc* can be more benefited from extensive testing, as sub-aspects usually depend on the behaviour of the super-aspect.

9.3.6.3 Empirical Data

Table 9.8 shows summary statistics of the *noc* metric and Figure 9.6 shows empirical data for *noc* metric both in aspects and classes. The x-axis shows the value of noc and the y-axis shows the relative frequency of each category in the projects.

Table 9.8: Summary statistics for *noc* values

Project Name	$\overline{noc}$ (Aspects)	σ	$\overline{noc}$ (Classes)	σ	$\overline{noc}$ rat.
AspectJ Design Patterns	0.4	0.7	0.02	0.1	19.5
AspectJ Examples	0.2	0.5	0.2	0.7	1.0
AspectJ Hot Draw	0.0	0.0	0.6	2.5	0.0
aTrack	0.3	0.7	0.2	0.8	1.4
Jakarta Cactus	0.0	0.0	0.4	0.8	0.0
Glassbox	0.5	1.4	0.7	1.4	0.7
GTalkWap	0.0	0.0	0.2	0.7	0.0
Infra Red	0.4	1.2	0.2	0.7	2.1
My SQL Connector J	0.0	0.0	0.3	1.1	0.0
Surrogate	0.7	1.2	0.2	0.5	4.2
Total	0.3	0.8	0.4	1.9	0.9

The majority of modules does not have children: in the selected projects 82% of the aspects and 87% of the classes do not have children. Aspects with more than one sub-aspect comprise 7.5% of the aspects and classes with more than one sub-class correspond to 6.8% of the total number of classes.

Six projects have a ratio between the mean value of *noc* for aspects and the mean value of *noc* for classes equals to zero or below one. On the other hand, four projects have more children in the aspects than in the classes: aTrack (1.38), Infra Red (2.13), Surrogate (4.22) and AspectJ Design Patterns (19.5). The value is the AspectJ Design Patterns project is high because the mean value for the *noc* of classes in the project is of only 0.02.

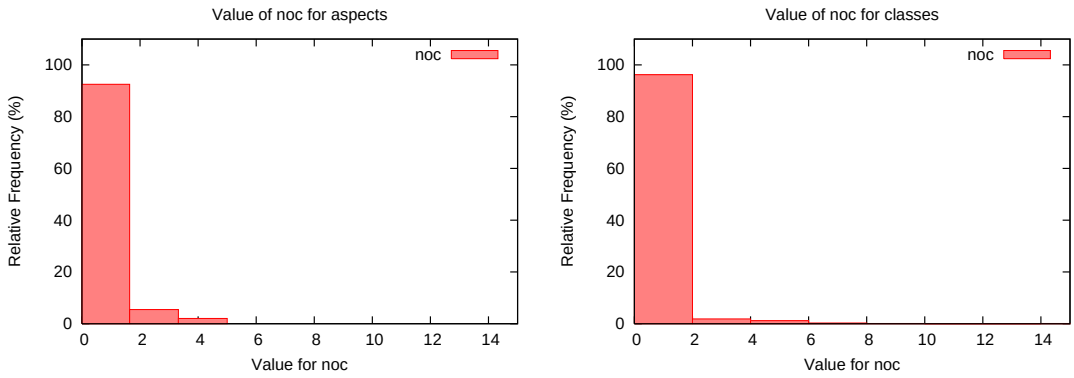

Figure 9.6: Value of *noc* for aspects and classes

9.4 Data Correlation

Correlation indicates the strength and direction of a linear relationship between two random variables (SNEDECOR et al., 1989). The correlation between the metrics was measured using the *gretl*[6] correlation algorithm. Tables 9.9 and 9.10 show the correlation values (r) for aspects and classes (rounded to two digits using unbiased rounding).

Table 9.9: Correlation coefficients between values for aspects

Correlation between values for aspects						
	dit	cae	cda	locc	noc	nom
dit	1.00	0.18	-0.25	-0.19	-0.06	-0.21
cae		1.00	0.17	0.26	0.06	0.26
cda			1.00	0.51	0.02	0.40
locc				1.00	0.22	0.87
noc					1.00	0.26
nom						1.00

Table 9.10: Correlation coefficients between values for classes

Correlation between values for classes					
	dit	cae	locc	noc	nom
dit	1.00	-0.07	0.02	-0.02	0.02
cae		1.00	0.00	0.00	0.01
locc			1.00	0.05	0.81
noc				1.00	0.13
nom					1.00

The correlation squared (r^2) (SNEDECOR et al., 1989) is used to help in the data interpretation. Correlation squared describes the proportion of variance in common between the two variables. High values of correlation squared appear only between the *locc* and *nom* metrics. This correlation squared is 0.76, which means that, across all the aspects in the sample projects, 76% of their variance on these two metric values is in common. Figure 9.7 shows the correlation between these two metrics in a scatter plot. There is also a small squared correlation between *locc* and *cda* (0.26) and between *nom* and *cda* (0.16).

[6] `http://gretl.sourceforge.net/`

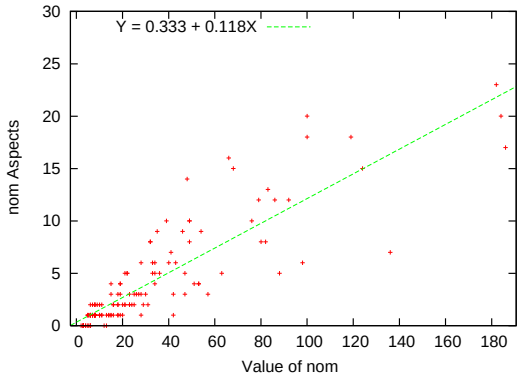

Figure 9.7: Correlation between *nom* and *locc*

These squared correlation values leads to the notion that there is a relationship between the size of aspects (in terms of the values of *locc* and *nom*) and the crosscutting degree of an aspect (*cda*) values. The remaining correlation values indicate very low correlation or no correlation at all between *dit* and the other metrics, *cae* and the other metrics and *noc* and the other metrics.

Table 9.9 also shows that the metrics for classes are not correlated, except for *nom* and *locc*, with a correlation of 0.81 (with a correlation squared of 0.66). Usually there is a balance between the number of lines of code per method in both aspects and classes. Note that correlation shows that a couple of values change together but does not necessarily imply causation, as the causes underlying the correlation may be indirect and unknown. Further investigation is needed to correlate these metrics with additional ones and to study the causes behind this correlation.

9.5 Using Metrics to Spot Shortcomings

This section shows a series of examples of classes and aspects, considering both high and low values for each metric and discuss the implications of such cases.

The main contribution of this chapter is an interpretation of collected empirical data, discussing the scope of values (minimum, maximum), comparing the values in aspects and in classes and examining variations between the metric values of the selected projects. A set of examples of high and low values for each of the selected metrics are shown and discussed, illustrating the value of these metrics on practical applications (including the correlation between the selected metrics).

9.5.1 Lines of Code

Classes with high values of $locc$ can be analysed and, if needed, broken into two or more classes. For example, the top ten classes in terms of $locc$ values, in the selected projects, are from the My SQL Connector J project. The $locc$ of these classes varies from 1317 to 4374 and they can be considered instances of the *Large Class* shortcoming (FOWLER et al., 1999).

The highest value of $locc$ for aspects in the selected projects is the *Tracer* aspect, in the My SQL Connector J project.

Inspecting this aspect, one can note that two private methods are not being used[7] and

[7]The *getStream* and *setStream* methods.

can be deleted (an occurrence of the *Unnecessary Code* shortcoming). The *Delete Method* refactoring pattern can be applied to such cases.

Also, there are two methods containing duplicated statements in their bodies (the *printEntering* and *printExiting* method). The *Extract Method* refactoring pattern (FOWLER et al., 1999) can be used to extract the common behaviour. The same case occurs with the *entry* and *exit* methods and with the *methods* and *constructors* pointcuts. Both are cases of code duplication. This aspect can be broken into two different aspects: one containing the core behaviour of tracing and other with the binding between this behaviour and the My SQL Connector J classes.

In the minimum case, there are occurrences of the *Unnecessary Code* shortcoming. For example, in the AspectJ Design Patterns project, there is an empty class named *Panel* and in the aTrack project, an empty aspect named *Observing*. Other empty classes in the selected projects are the *Sorter* and *ButtonCommand2* classes in the AspectJ Design Patterns, the *HTMLTextAreaFigure* inner-class named *InvalidAttributeMarker* in the AspectJ Hot Draw project and the *MockMethodTestCase* inner-class named *TestException* in the Surrogate test framework. Empty classes can be considered a kind of unnecessary code.

Classes with low values of $locc$ can also bring shortcomings. There are 25 classes and aspects with a $locc$ value below five and 129 below six, for example. The developer can inspect these small classes and aspects to evaluate if their existence is justified or if they can be merged with existing classes.

9.5.2 Number of Operations in Module

Two classes with the highest values for the wom metric in the selected projects were analysed: *Connection* and *ConnectionProperties* (from the My SQL Connector J project). The *Connection* class has an inner class, named *UltraDevWorkAround*, with 154 methods.

Inspecting the *ConnectionProperties* class, the first thing to note is that it has a lot of inner classes. Figure 9.8 shows the *ConnectionProperties* class with the attributes and methods compartments hidden to better visualize both the inner classes and two of its sub classes: *PropertiesDocGenerator* and *DocsConnectionPropsHelper*.

The dependency between *ConnectionProperties* and the selected sub-classes is very weak. The sub-classes have only a *main* method. As the source code to both classes is available, the *PropertiesDocGenerator* class (Listing 9.1) was inspected. This class does not need to extend the *ConnectionProperties* class.

Listing 9.1: `PropertiesDocGenerator` class

```
1 public class PropertiesDocGenerator
2   extends ConnectionProperties {
3   public static void main(String[] a) throws SQLException {
4     System.out.println(new PropertiesDocGenerator().
        exposeAsXml());
5   }
6 }
```

Listing 9.2 shows that the inheritance dependency can be removed and an instance of the *ConnectionProperties* class (line 3) can be created instead.

Listing 9.2: `PropertiesDocGenerator` class Modified

```
1 public class PropertiesDocGenerator{
2   public static void main(String[] a) throws SQLException {
```

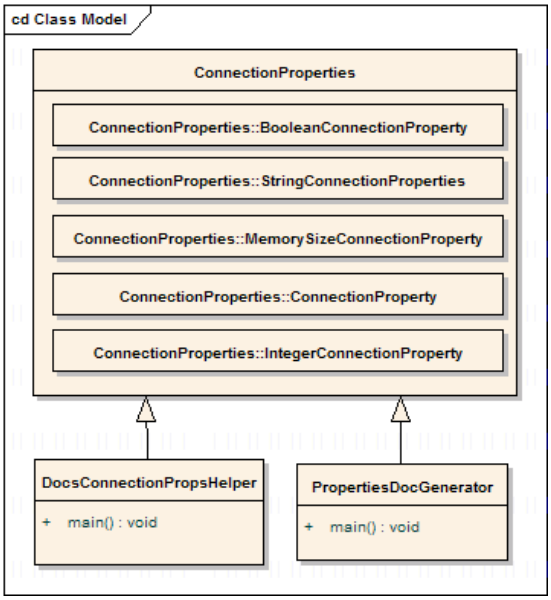

Figure 9.8: Class diagram for the *ConnectionProperties* class and some of its sub-classes

```
3         System.out.println(new ConnectionProperties().exposeAsXml
              ());
4     }
5 }
```

Also, the *DocsConnectionPropsHelper* is equal to the *PropertiesDocGenerator* class, except for its name. These classes present two shortcomings: *code duplication* and *lazy classes*. None of them are being used by other classes and they can be both deleted (applying the *Delete Class* refactoring pattern).

If they are being used indirectly, through reflection, only one of them is needed. Also, the inner classes are quite big and can be extracted to new classes. This can reduces the size of the *ConnectionProperties* class and its complexity.

In terms of high values of wom in aspects, consider the *ExecutionTracer* aspect in the Glassbox project, for example. It has 20 operations dealing with trace printing, pattern matching and advising trace points. It can be refactored into a set of aspects dealing each one with a different concern. Large aspects can be benefited from the use of a guideline named *one concern per aspect* (Appendix B), which proposes that each aspect should encapsulate a single concern.

Low values of wom in classes appear, for example, in the *Version* class (Listing 9.3) of the Jakarta Cactus project. This class has only a constant field (line 2). This constant can be moved to another class or stored in a resource bundle.

Listing 9.3: `Version` class

```
1 public class Version {
2   public static final String VERSION = "@version@";
3 }
```

Another class with a low wom value is the *EscapeProcessorResult* class, from the

My SQL Connector J project (Listing 9.4). Usually, it is interesting to encapsulate the access to the attributes using accessors. For example, the *escapedSql* attribute (line 3) is used by six different methods in three classes. The developer can use the *Encapsulate Attribute* refactoring pattern (FOWLER et al., 1999) to provide a *getEscapedSql* and a *setEscapedSql* methods to access the protected data. This allows the structure of the *escapedSql* to be changed (from a *String* for a *StringBuffer*, for example), decoupling the *EscapeProcessorResult* class from the classes that use it.

Listing 9.4: `EscapeProcessorResult` class

```
1 class EscapeProcessorResult {
2     boolean callingStoredFunction = false;
3     String escapedSql;
4     byte usesVariables = Statement.USES_VARIABLES_FALSE;
5 }
```

Low values of wom in aspects occur, for example, in the abstract aspect *TemplateOperationMonitor*, in the Glassbox project. This aspect has four pointcuts that can be extended by sub-aspects but does not have any associated behaviour. This aspect can be seen both as an occurrence of the *Lazy Aspect* shortcoming and as an occurrence of the *Speculative Generality* shortcoming (FOWLER et al., 1999; PIVETA et al., 2005). Other aspects with too few responsibilities to justify its existence include the *AtrackLogManager* and the *AtrackExceptionHandling* aspects, in the aTrack project.

For example, the *AtrackLogManager* (Listing 9.5) only defines a declare parents statement (line 2), that can be moved to another aspect that deals with logging.

Listing 9.5: `AtrackLogManager` aspect

```
1 public aspect AtrackLogManager {
2   declare parents: org.atrack..* implements Loggable;
3 }
```

9.5.3 Depth of Inheritance Tree

In the aTrack project, there are four classes with a dit of five or six. All these classes represent Java exceptions, with the following hierarchy:

```
1 -Object
2   - Throwable
3     - Exception
4       - RuntimeException
5         - AtrackException
6           - PersistenceException
7           - ControllerException
8           - ModelException
9             - EntityNotFoundException
```

As user defined exceptions in Java usually extend from *RuntimeException* or from one of its sibling classes, it is expected that they have a dit of four or more. In this case, however, one of the classes in the inheritance tree, the *AtrackException* class, is used only in the *LoginAction* class (Listing 9.6 - line 8):

Listing 9.6: `LoginAction` class

```
1 public class LoginAction extends Action {
```

```
2     ...
3     private Subject authenticate(String username, String
          password){
4       ...
5       try {
6         lc.login();
7       } catch (LoginException e) {
8         throw new AtrackException(e);
9       }
10      return lc.getSubject();
11    }
12  }
```

This class is an unnecessary *middle man* (FOWLER et al., 1999). The *PersistenceException*, *ControllerException* and *ModelException* can inherit from *RuntimeException* and the *AtrackException* can be deleted, as it is used in only one place, which can be changed to a direct reference to the *RuntimeException* class.

In the selected projects, the maximum value for the dit of aspects is three. There are no problems associated with such values for dit in aspects. In fact, only two aspects actually have a dit equal to three: the *AbstractXmlCallMonitor* and the *XMLParsingMonitor* aspects in the Glassbox project. Further inspection in these two aspects does not show any misuse of inheritance.

Low values of dit in classes are quite common. It is only a problem if the class is bloated with a lot of responsibilities and the use of inheritance can alleviate the problem or if the class is an occurrence of the *Lazy Class* shortcoming and has few responsibilities in the overall design.

As the inheritance in aspects plays a slightly different role that the inheritance of classes, the values of dit of aspects are expected to be lower than the dit of classes. Usually, inheritance mechanisms are used to decouple the behaviour defined in a super-aspect with the concrete join points specified in the sub-aspects. Examples of such use of inheritance are representative of a design guideline named *Use abstract aspects* (Appendix B), which states that the developer should design towards abstract aspects, whose behaviour is defined completely by its advices, and its relationship with classes or other aspects is accomplished by specialization.

The use of abstract aspects can help in developing more reusable aspects, by postponing implementation decision and leaving the definition of concrete pointcut definitions to the sub-aspects. Also, the behaviour defined in abstract aspects can be reused to different target applications. Each application can create sub-aspects that capture the specific points that will activate the aspect behaviour.

For example, in the *Observer* pattern (HANNEMANN; KICZALES, 2002) (in the AspectJ Design Patterns project) there is a *ScreenObserver* aspect (Listing 9.7) that extends the reusable abstract aspect *ObserverProtocol* (line 1) and defines that both roles (*Subject* and *Observer*) will be played by the *Screen* class (lines 2 and 3). It also defines when the subject state changes (line 4) and what should be done to update the observers (lines 5-7). This example defines an abstract aspect implementing the logic for the *Observer* pattern and leaves for the sub-aspects the task of binding the *Subject* and *Observer* roles and the changes in the *Subject* with the classes that will play these roles.

Listing 9.7: `ScreenObserver` aspect

```
1  public aspect ScreenObserver extends ObserverProtocol{
```

```
2     declare parents: Screen implements Subject;
3     declare parents: Screen implements Observer;
4     pointcut subjectChange(Subject sub):
5       call(void Screen.display(String)) && target(sub);
6     void updateObserver(Subject sub, Observer obs) {
7       ((Screen)obs).display("Updated");
8     }
9   }
```

9.5.4 Number of Children

High values of noc can be seen in classes that are highly reused through inheritance. In the AspectJ Hot Draw project, for example, the *AbstractCommand* class has 33 children and is the base class for new *Command* classes. Other examples of classes with a high value of noc in the AspectJ Hot Draw framework are the *UndoableAdapter* (with 24 children), the *AbstractTool* (with 15 children) and the *ResizeHandle*, with eight children.

Aspects with high values of noc usually implement the basic behaviour of a concern and use abstract pointcuts to define a contract that the sub-aspects must fulfil.

Consider the *InfraREDBaseAspect* aspect (Listing 9.8), from the Infra Red project, for example. This aspect tracks the time spent by a method call and updates a set of statistics. It defines an abstract pointcut and an abstract method as hooks that are overridden by the sub-aspects, binding the application classes with the time tracking behaviour. The *condition* abstract pointcut (line 2) specifies the condition based on which monitoring is performed and the *getApiType* method (line 3) gets the type (Session Bean/Entity Bean/JDBC) of an API.

Listing 9.8: `InfraREDBaseAspect` aspect

```
1  public abstract aspect InfraREDBaseAspect {
2    public abstract pointcut condition();
3    public abstract String getApiType();
4    Object around() : condition(){
5      ...
6      final String apiType = getApiType();
7      // Time tracking statements
8    }
9  }
```

The *InfraREDBaseAspect* aspect has four sub-aspects: *EntityBeanAspect*, *SessionBeanAspect*, *StrutsAspect* and *WebAspect*. The *WebAspect* aspect (Listing 9.9), for example, overrides the *condition* pointcut (line 2) to define which join points are affected by the *InfraREDBaseAspect* behaviour and the *getApiType* (line 5), to specify the type of API used.

Listing 9.9: `WebAspect` aspect

```
1  public aspect WebAspect extends InfraREDBaseAspect {
2    public pointcut condition():
3      execution (public * HttpServlet+.*(..)) ||
4        execution (public * Filter+.*(..));
5      public String getApiType() {
6        return "Web";
7      }
```

```
8     }
```

Low noc values in classes and aspects are commonplace. In fact, in the selected projects, nearly 87% of all the aspects and classes do not have sub-classes or sub-aspects.

9.5.5 Crosscutting Degree of an Aspect

Examples of aspects with high values of cda include the *Tracer* aspect from the My SQL Connector J project (cda = 78), the *LogAspect* from the Jakarta Cactus project (cda = 68), the *AtrackLogManager* (cda = 39) from the aTrack project and the *QueueStateAspect* (cda = 38) from the AspectJ Design Patterns project.

Consider the *Tracer* aspect (Listing 9.10), for example. This aspect has a pointcut named *methods* that defines the join points using wildcards and package information, instead of simply listing all the affected points. It ensures that every method execution within a set of packages is traced.

Listing 9.10: `Tracer` aspect

```
1  public aspect Tracer {
2     pointcut methods(): execution(* *(..))
3     && within(com.mysql.jdbc.* )
4     && within(!com.mysql.jdbc.trace.*)
5     && within(!com.mysql.jdbc.log.*)
6     && within (!com.mysql.jdbc.Util);
7      ...
8  }
```

The *QueueStateAspect* (Listing 9.11) defines behaviour according to class initialisation. The after advice with the *initialization(new()) && target(q)* pointcut expression affects 34 classes (line 2). If new classes are added to the system, they will be automatically affected by the aspect.

Listing 9.11: `QueueStateAspect` aspect

```
1  public aspect QueueStateAspect {
2    after(Queue q): initialization(new()) && target(q) {
3      q.setState(empty);
4    }
5     ...
6  }
```

The *ErrorHandling* aspect, in the Glassbox project, affects 33 classes using a composite pointcut, defining the affected join points using nine separated predicates (one for each set of points).

Aspects with low cda values should be inspected to evaluate if they can be converted to classes or merged with other aspects. The refactoring pattern *Convert Aspect to Class* can be applied to transform the aspect to a class. *Inline Aspect* is another refactoring pattern that can be applied to eliminate an aspect. An alternative is to apply *Move Attribute*, *Move Method*, *Move Advice*, or *Move Inter-Type Declaration* refactoring patterns to empty the aspect, and later apply the *Delete Aspect* refactoring pattern.

Sometimes the aspects with low cda values extend other aspects in a similar way that the *Template Method* design pattern (GAMMA et al., 1995) is implemented in object-oriented software.

Consider, for example, the *ExampleProjectCalls* aspect (Listing 9.12) in the Surrogate project. It extends the *SurrogateCalls* aspect (line 5), that defines an abstract pointcut named *mockPointcut* (line 6) and an advice that implements a certain behaviour each time the *mockPointcut* join points are reached (line 7).

Listing 9.12: `ExampleProjectCalls` aspect

```
1  aspect ExampleProjectCalls extends SurrogateCalls {
2    protected pointcut mockPointcut() : (call (java.io.*Reader.
       new(..)) ||
3      call(* java.lang.System.currentTimeMillis())) ;
4  }
5  public abstract aspect SurrogateCalls {
6    protected abstract pointcut mockPointcut();
7    Object around() : mockPointcut(){
8      ...
9    }
10 }
```

Other examples of aspects that implement this aspect-oriented version of the *Template* design pattern, and have a low value for cda include the four *InfraREDBaseAspect* children in the Infra Red project (*EntityBeanAspect*, *SessionBeanAspect*, *StrutsAspect* and *WebAspect*) and the *ColorObserver* and *RequestCounting* aspects in the AspectJ Design Patterns project.

Another use of the cda metric is to spot occurrences of the *Lazy Aspect* shortcoming. In the AspectJ Design Patterns project, for example, the *Lazy Aspect* shortcoming appear in four aspects: *StrategyProtocol*, *MementoProtocol*, *FlyweightProtocol* and *CompositeProtocol*. These aspects do not have any crosscutting members and can be converted to classes using the *Convert Aspect to Class* refactoring pattern. Whenever an aspect does not have members implementing crosscutting concerns a class can (and should, if possible) be used instead. One *lazy aspect* was detected in the Glassbox. The *AbstractResourceMonitor* aspect does not have crosscutting members, but it cannot be converted to a class because it extends the *AbstractRequestMonitor* aspect (in AspectJ, classes cannot extend aspects).

9.5.6 Coupling on Advice Execution

High values of cae can be an indicative of the *Aspect Interaction* shortcoming. The developer should focus the search for aspects interactions on the modules with the highest values for this metric. Table 9.11 shows the number of modules with $cae > 1$ in the selected projects. Table 9.12 shows the same information considering a $cae > 2$. Classes with a cae value higher than one can have interaction issues. The probability of having interaction problems is higher in modules with high values for the cae metric.

The aTrack and the AspectJ Examples projects have several classes with values of cae higher than two. Note that this fact is not a problem itself, but the classes should be inspected to detect occurrences of the *Aspect Interaction* shortcoming. If there are interactions (i.e. two or more advices are affecting the same joinpoint), there are the need to revisit the precedence of the aspects and adjust it if needed. This can be performed by applying a refactoring pattern named *Add Aspect Precedence*.

Consider, for example, the *LoginAction* class (Listing 9.13) in the aTrack project. This class has a cae value equals to ten. It means that ten different aspects affect this class. The

Table 9.11: Number of modules with $cae > 1$

Project	# of modules
aTrack	51
ajExamples	22
glassbox	9
ajDesignPatterns	8
GTalkWAP	2
ajHotDraw	2

Table 9.12: Number of modules with $cae > 2$

Project	# of modules
aTrack	47
ajExamples	11
glassbox	3

class has only two methods and its behaviour is heavily influenced by the aspects. It is difficult to see if the interaction of all the aspects affecting this class is correct, in the right order or even how they affect the behaviour of the class.

Listing 9.13: `LoginAction` class

```
1  public class LoginAction extends Action {
2    public ActionForward execute (ActionMapping mapping,
3      ActionForm form, HttpServletRequest request,
         HttpServletResponse response)
4      throws Exception{
5        LoginForm loginForm = (LoginForm)form;
6        ...
7        return mapping.findForward
8          (Consts.SUCCESS_REDIRECT);
9    }
10   private Subject authenticate(String username,
11     String password){
12     ...
13   }
14 }
```

The class is advised by three advices defined in the *ExecutionTracer* and the *ExceptionHandling* aspects, there are three parents declarations from the *PersistenceControl* and the *AtrackLogManager* aspects. Thirteen methods are added by inter-type method declarations from the *LogManager* aspect. The *execute* method (line 2) is advised by twelve different advices, defined in eight different aspects and one exception is softened by the *ErrorHandling* aspect. The *authenticate* method (line 10) is advised by three advices defined in two different aspect.

It is difficult to reason about the resulting behaviour of the aspects that affect this class. The development environment can help to show these occurrences of the *Aspect Interaction* shortcoming. However, it is currently an open issue how this is modelled and implemented in a way to ensure that the behaviour is correct. Once the correct behaviour is defined, the developer can adjust the correct precedences with the refactoring patterns

Add Aspect Precedence and *Remove Aspect Precedence*.

Low values of cae are common and do not represent any issues in terms of complexity, reusability or maintainability. A cae value of zero denotes that the class or aspect is not affected by any aspects.

9.5.7 Discussion

Metrics adapted from (CHIDAMBER; KEMERER, 1994) and metrics specifically tailored for aspect-oriented software can be used to evaluate software in the presence of aspects in several ways. Metrics can be used as indicators of quality, used to measure quality attributes, such as reusability or modularity, for example. Also, they can be used to detect problems that can appear in the software. This section provides a summary of how the usage guidelines and the examples discussed can provide insights on how each metric can be used to spot shortcomings in aspect-oriented software.

The *locc* metric can be used in combination with other metrics as an indicative of effort, complexity, productivity and cost. In the selected projects, the majority of classes and aspects have low values of *locc*, showing probable design efforts attempting to improve the comprehensibility and to reduce the number of defects per module. More specifically the *locc* metric can be used to spot:

- *Large Operations:* The locc metric in combination of *nom* is used to evaluate the size of operations. High *locc* values with low values of *nom* can show large methods, large advices or large inter-type method declarations.

- *Unnecessary use of Aspects:* Classes with high *locc* values and with low *cda* values can denote that the aspect has state or behaviour that is not dealing with crosscutting concerns.

- *Aspect Interactions:* Aspects or classes with low *locc* values and high *cae* values can suffer from aspect interactions, where more than one aspect affects the same join points at the same time.

- *Large Modules:* High values of *locc* in classes and aspects can denote large classes or large aspects. Those modules should also be inspected for code duplication, for unused operations and attributes and for the encapsulation of more than one concern per module.

- *Lazy Modules:* Classes or aspects with very low values of *locc* should be inspected to evaluate if they have enough responsibilities to exist at all.

The *nom* metric can indicate how much effort is needed to develop an aspect or a class. Modules with high values for *nom* are likely to be more application specific, with a lower reusability. The metric can also be used to detect a set of cases:

- *Lazy Modules:* Modules with low values of *nom* can be evaluated to see if they have enough responsibilities to be first class entities of a system or if it is better to merge them with other modules;

- *Large Modules:* Modules with high values of *nom* can be large modules, with several concerns being encapsulated or having a large number of inner classes or unrelated operations, for example;

- *Refused Bequest:* Modules with high values of *nom* are more likely to suffer from this shortcoming, in which sub-classes or sub-aspects inherit methods that are not used.

The values for the *nom* metric are highly correlated to those of *locc*. As happens with the values of *locc*, the *nom* of classes is higher than in the aspects. Around 80% of modules have a maximum of ten operations, but several modules with high values of *nom* can be found in practice.

The values of *dit* in classes are usually higher than in aspects, as in AspectJ the aspects have a limited inheritance mechanism. The *dit* metric can be used to:

- *Measure Complexity:* Modules with high values of *dit* are usually more complex and more project specific, limiting reuse.

- *Detect Misuse of inheritance:* Modules with high *dit* values should be inspected to search for misuse of inheritance, i.e. should be analysed to evaluate if too much emphasis is given to inheritance.

The *noc* metric can indicate how many modules use inheritance as a reuse mechanism. In the selected projects, around 85% percent of the modules do not have children. The *noc* of an aspect or class is usually used to spot:

- *Indicatives of Reuse:* Modules with several children are likely to be extensively reused. The developer should look for sub-classes or sub-aspects that do not use effectively the composing elements of their super-classes or super-aspects;

- *Important Modules:* Sometimes, modules with several children are important modules of a project. These modules can be tracked and analysed for any inheritance misuses.

The *cda* metric is used to measure the influence of an aspect in other modules and can be used to:

- *Evaluate Usefulness of Aspects:* Aspects with high values of *cda* are usually more valuable, as the equivalent object-oriented modularisation would be scattered over several modules.

- *Find Lazy Aspects:* Aspects with a low *cda* can be inspected to see if the behaviour and state encapsulated by the aspect can be moved to or merged with other modules.

Aspects that deal with global policies, such as logging, tracing or authentication are more likely to have high values of *cda* than other aspects. In the selected projects, the majority of aspects does have low values for *cda* and some of them can be seen as lazy aspects.

The *cae* metric represents the number of aspects that affects a given module and is mainly used to detect:

- *Aspects Interaction:* The higher the value of *cae* of a module, the higher the probability of having more than one aspect affecting the same join points of this module;

- *Affected Modules:* The modules that have a *cae* different than zero are those that are affected by some aspect. These modules have to be treated more carefully, as modifications in the affected module can potentially affect the aspect behaviour.

In the selected projects, classes are more affected by aspects than the aspects themselves. Furthermore, the values of this metric are quite low, except for a few modules (there are modules, for example with a *cae* of ten). Low values of *cae* are common and do not represent any issues in terms of quality attributes.

9.6 Related Work

Analysis of empirical data is a straightforward way to investigate the benefits and disadvantages of software properties. This is also the goals of some related work in the literature. This section summarizes the main characteristics of these related works and compares them with the work of this chapter.

Baxter et al. (BAXTER et al., 2006) analysed a corpus of Java programs to provide information about the typical values of metrics in Java programs, aiming at understanding the relationship among Java classes and objects. The work is this chapter differs from theirs in the sense that this chapter discusses the shape of aspect-oriented programs and focus on the formal definition of the metrics, and empirical data rather than verifying if the distribution function of the metrics obeys power laws (as in their work).

Zhao (ZHAO, 2002) propose a set of metrics to aspect-oriented software to quantify the information flow in aspect-oriented programs. He also discusses a set of metrics to coupling in aspect-oriented software (ZHAO, 2004) and a set of metrics to compute the cohesion of aspect-oriented software (ZHAO; XU, 2004). His metrics are also formally defined and evaluated according to a set of well-defined criteria. The main difference is that this thesis deals with a different set of metrics, regarding size, inheritance and aspect-specific coupling metrics and it shows typical values for aspect-oriented software, data interpretation and the correlation of the metrics.

Other authors propose metrics for aspect-oriented software (SANTANNA et al., 2003; ZAKARIA; HOSNY, 2003; TONELLA; CECCATO, 2004). They defined the metrics informally and do not conduct analytical evaluation of the proposed metrics, neither empirical data to describe the common characteristics of aspect-oriented software. This thesis provides a formal definition, empirical data and analytical evaluation of the metrics (Appendix C).

Bartsch and Harrison (BARTSCH; HARRISON, 2008) deal with the empirical validation of aspect-oriented coupling measures as indicators of maintainability of aspect-oriented software and with the validity of those metrics in terms of a set of theoretical principles (BARTSCH; HARRISON, 2006). They state that there is a weak correlation between a set of coupling and size metrics with the maintenance effort between different versions of an application. Although there is the need for further research to validate coupling metrics for aspect-oriented software as indicators of maintainability, their work is a first step on the validity of coupling metrics for aspect-oriented software. Both of their works are in an initial stage and are grounded on an informal basis. The work of this thesis complements their works by providing formal definitions of metrics, analytical evaluation, usage scenarios, empirical data and interpretation to two of the five coupling metrics that those authors discuss (*cae* and *cda*).

9.7 Conclusions

In this chapter, a set of contributions to the use of metrics for aspect-oriented software are provided. More specifically, it provides for a set of six metrics: a formal definition

and a set of usage scenarios and (ii) an interpretation of collected empirical data, and the correlation between metrics. Appendix B shows an analytical evaluation of the metrics against established criteria for validity.

The use of formal definitions for the metrics helps to understand the metrics more clearly and unambiguously, making it easier to ensure that the computation of the metric values can be done in a repeatable fashion. It can also facilitate the automation of the metrics collection process.

The set of usage scenarios can show the developers how the metrics can be used to detect shortcomings in existing software artefacts. For example, the metrics evaluated in this chapter can be used to show the occurrence of several cases, as follows. The *locc* metric can be used to spot large operations, unnecessary use of aspects, aspects interaction and large modules. The *nom* metric can be used to detect lazy and large modules, refused bequests, influence of aspects and how affected is a given module. The *dit* metric is used primarily to measure complexity, misuse of inheritance and the occurrence of project specific aspects. The *noc* metric can indicate the degree of reuse of a given module and to spot key modules of a project. The *cda* metric is used to evaluate the usefulness of an aspect and to find lazy aspects. The *cae* metric is mainly used to show if the module is affected by aspects and to show the possibility of aspect interaction scenarios.

The data interpretation shows typical values of aspects and classes in aspect-oriented software. The provided histograms can be used to compare the metric values for an aspect-oriented project with the set of open source projects used in this chapter.

The correlation between the metrics explains how certain metrics change together and how they can be combined and used to evaluate aspect-oriented software. It is shown that the metrics both for aspects and classes are not correlated, except for *locc* and *nom* (high correlation) and between *locc* and *cda* and *nom* and *cda* (small correlation), but can nevertheless be combined to show cases in which the software can be improved.

Further research is needed to assess and evaluate other metrics for aspect-oriented software, such as metrics for coupling and cohesion, including: coupling on intercepted modules (cim), coupling on method call (cmc), coupling on field access (cfa), response for a module (rfm) and lack of cohesion in operations (lco). More details about these specific metrics can be found in Ceccato and Tonella (CECCATO; TONELLA, 2004). Future work can also focus on the analysis of Zhao's metrics (ZHAO, 2002) to spot shortcomings on aspect-oriented applications.

10 CONCLUSION

This thesis presents a set of contributions to improve the search for refactoring opportunities in aspect-oriented and object-oriented software applications. It includes approaches to select and rank refactoring patterns according to a quality model, to search for refactoring opportunities in software artefacts, to reduce the number of refactoring sequences, and to evaluate the effects of refactoring on software quality. The thesis also includes a catalogue of shortcomings in aspect-oriented software and a case study of metrics for aspect-oriented software. This chapter is organised as follows. Sections from 10.3 to 10.7 summarise the contributions of this thesis, how each of them was evaluated, which tool support was developed and how the contributions are inter-related. Section 10.8 describes the main areas for future work regarding the main ideas of this thesis.

10.1 A Discipline for Refactoring

This thesis proposes a discipline for guiding the developers when performing refactoring activities. It provides a set of activities, roles, artefacts, and tool support for refactoring. The discipline is divided into two stages: preparation and search.

The preparation activities are organised as follows. In the first step, the developers select a quality model for the software application being developed. The next step is to select refactoring patterns for which refactoring opportunities will be searched. If needed, a ranking of refactoring patterns according to the quality model can be created. The next step is the creation or selection of heuristic rules.

After the quality models, the refactoring patterns and the heuristic rules are selected, the developers can search for refactoring opportunities. First, they specify the search scope, which includes the selection of which classes, and which packages will be searched, and the definition of whether searching for refactoring opportunities in software elements for which refactoring patterns were already applied. Then, the developers set the number of levels of successive refactoring for the search (see Chapter 8).

Having the search scope defined, the next steps are to search for software elements matching the scope, to compute the values for the heuristic rules, and to create a list of refactoring opportunities. The developers can evaluate the advantages of each refactoring pattern application either quantitatively (using impact functions) or qualitatively (by seeing the changes that the refactoring pattern application can have).

The developers can mark a set of refactoring patterns to be applied and move to the next activity: application of refactoring patterns. In this activity, the developers provide additional parameters needed by each refactoring pattern, apply the refactoring patterns, test the affected modules and, if needed, undo some of the applications of refactoring patterns. These search activities can continue until the developers decide to stop the search

for refactoring opportunities.

A discipline containing activities for refactoring can help the developers to organize the refactoring process, focusing on reducing the required efforts and improving the effective results. This is accomplished by improving a particular software module on a set of chosen quality attributes, and by applying a set of selected refactoring patterns that contribute to the improvement of the selected quality attributes on a set of selected elements - chosen by their chances of being improved by the selected patterns.

This thesis provides tool support for ranking refactoring patterns, for searching for refactoring opportunities, and for computing the effects of refactoring on software quality. Tool support for the creation of quality models and the application of refactoring patterns can be provided by third party tool vendors (such as the support for the application of refactoring patterns in current IDEs).

The ranking of refactoring patterns using AHP, according to a set of quality attributes of a piece of software, is described in Chapter 4. Chapter 5 describes how to automatically detect typical shortcomings which occur in aspect-oriented software. Chapter 6 shows how metrics can be grouped together as heuristic rules and how they can be used to prioritise refactoring opportunities. Chapter 7 describes how impact functions can be created and used to quantitatively evaluate the quality of the software application being developed. Chapter 8 describes an approach to reduce the number of sequences of refactoring patterns, by proving ways to create a set of initial sequences and to create simplification rules to reduce the number of sequences.

10.2 A Method for Ranking of Refactoring Patterns

This thesis describes how to rank a set of refactoring patterns according to their contribution to the required quality attributes of a piece of software. The refactoring patterns are ordered by their expected contribution to a set of quality attributes and the quality attributes are ordered by their relative importance in the current project. To accomplish this ordering process, a multi-criteria decision method named Analytical Hierarchy Process (AHP) is used. This ranking of refactoring patterns can be used to focus the refactoring effort on the most promising refactoring patterns to the software application being developed or maintained.

AHP provides mechanisms to express the relationship between quality attributes and refactoring patterns and quality attributes between them. A ranking can then be computed and used for the selection of refactoring patterns. The proposed approaches are adaptable: the quality attributes, the refactoring patterns, and the weights can be changed. In each change, the new ranking can be computed automatically.

The proposed approach was instantiated for a detailed example of the construction of a ranking composed of three quality attributes and four refactoring patterns. This example shows, step by step, how to specify pairwise comparisons for the quality attributes, how to specify pairwise comparisons for refactoring patterns according to the quality attributes, and how to compute an overall ranking of refactoring patterns according to the selected quality attributes.

Tool support was developed to help in the creation of pairwise comparisons for quality attributes and refactoring patterns using AHP, including the components needed to compute the ranking of quality attributes, the ranking of refactoring patterns and the overall ranking (of refactoring patterns according to the quality attributes).

Such approach for ranking refactoring patterns can be used to focus the search for

refactoring opportunities on those refactoring patterns that are more likely to improve the selected quality attributes. The usage guidelines of the metrics described in Chapter 9 can be used together with the impact functions described in Chapter 7 to improve the accuracy and precision of the generated ranking (by improving the pairwise comparisons).

10.3 An Approach to Search for Refactoring Opportunities

This thesis proposes an approach to search for refactoring opportunities in software artefacts. It focuses on identifying and prioritising refactoring opportunities, aiming at maximizing the quality attributes the developers are interested in. The approach uses heuristic rules to evaluate the software artefacts according to a selected quality model, and qualitative analysis to evaluate the trade-offs for each opportunity.

The advantage of using this approach is that the developers can automatically detect occurrences of shortcomings and software elements with low values for heuristic rules, which are specified to quatitatively evaluate the quality attributes of a software application. Also, the application of refactoring patterns is suggested for each refactoring opportunity, which allows the developers to improve the software application through refactoring.

To evaluate and show the applicability of the proposed approach in a practical setting, two case studies are provided. The first case study was conducted in an object-oriented software application, in which a heuristic rule (composed of two quality attributes and a set of metrics) and two refactoring patterns were used to search for refactoring opportunities. The identified and prioritised refactoring opportunities were evaluated as well as the resulting classes after the application of a set of refactoring patterns. The second case study shows the search for occurrences of shortcomings in a set of aspect-oriented software applications using a set of heuristic rules for AspectJ.

Tool support includes an AST-based tool for Java and AspectJ, developed in the context of this thesis, as a plug-in for the Eclipse development environment. This tool includes a set of algorithms to search for refactoring opportunities in AspectJ programs, using AST visitors. The aopmetrics (STOCHMIALEK, 2009) tool was used to collect the metrics and store them in a Postgresql database. SQL queries were used to compute the heuristic values and generate the prioritised set of refactoring opportunities. The eclipse refactoring tool was used to apply the refactoring patterns. Chapter 3 lists additional tools for the search and prioritisation of refactoring opportunities.

The search and prioritisation process starts with a set of refactoring patterns. Such set can be the best ranked refactoring patterns obtained by the approach described in Chapter 4. The metrics presented in Chapter 9 can be used to quantitatively express the quality attributes that the developers are aiming for. The impact functions created using the approach proposed in Chapter 7 can be also used to prioritise the detected refactoring opportunities. This approach can also be used to identify and prioritise opportunities to apply refactoring sequences (described in Chapter 8). More on the meta-model of such approach, the roles and artefacts needed, please refer to Chapter 3.

10.4 A Catalogue of Shortcomings in Aspect-Oriented Software

This thesis describes a catalogue of shortcomings that arise in aspect-oriented systems, suggests refactoring patterns that can be used to remove or minimize them, and provides tool support for automatically detecting occurrences of these shortcomings in software

applications. It extends other works that aim at defining catalogues of shortcomings in object-oriented code (FOWLER et al., 1999) and shortcomings in aspect-oriented code (MONTEIRO; FERNANDES, 2005a).

The main advantages are to help in the development of automated tools to identify typical shortcomings in software applications and to suggest refactoring patterns to minimise or remove those shortcomings. The catalogue of shortcomings can also provide insights to the definition of design guidelines (such as the ones described in Appendix B), which aim at preventing the insertion of such shortcomings in the software application.

Examples retrieved from well-known aspect-oriented programs are provided for each shortcoming described in the catalogue. Chapter 6 shows a case study using a set of heuristic rules that can be used to detect the occurrences of shortcomings in aspect-oriented software applications.

The number of shortcomings can be reduced using the approach described in Chapter 4. In this case, only the shortcomings with an associated refactoring pattern in the ranking are included in the search. The metrics described in Chapter 9 can be used to define heuristic rules, both in aspect-oriented and object-oriented software programs. Impact functions (described in Chapter 7) can be associated with each opportunity for refactoring found by a detection tool. This way, the developer can evaluate both quantitatively and qualitatively if the suggested refactoring pattern application is advantageous or not.

10.5 Metrics for Evaluating the Quality of Aspect-Oriented Software

This thesis provides a set of contributions to the use of metrics for aspect-oriented software. More specifically, it provides, for a set of six metrics: (i) a rigorous definition and a set of usage scenarios, (ii) an interpretation of collected empirical data, including the correlation between these collected metrics, and (iii) an analytical evaluation of the metrics against established criteria of validity (described in Appendix C).

Four of the selected metrics (*locc*, *nom*, *dit* and *noc*) can be used to measure size and use of inheritance. The other two metrics (*cda* and *cae*) show how many modules an aspect affects and also how many aspects affect each module. These two coupling metrics provide basic information about the influence of aspects in the overall design.

The use of formal definitions helps to understand the metrics more clearly and unambiguously, making it easier to ensure that the computation of the metric values can be done correctly. The set of usage scenarios can show the developers how the metrics can be used in practice to detect shortcomings in existing software artefacts. The data interpretation shows typical values of aspects and classes in aspect-oriented software. The provided histograms can be used to compare the metric values for an aspect-oriented project with the set of open source projects used in Chapter 9. Furthermore, the correlation between the metrics explains how certain metrics change together and how they can be combined and used to evaluate aspect-oriented software.

No specific tool support was developed regarding the metrics in the case study. Aopmetrics was used to collect the metrics (STOCHMIALEK, 2009). Gnuplot was used for the generation of graphs (`http://www.gnuplot.info/`), and a custom generation program was developed in Java to generate the statistical data. Correlation was computed using the Gretl tool (`gretl.sourceforge.net/`).

The metrics used in the case study can be used for the definition of pairwise comparisons between refactoring patterns according to quality attributes. Such pairwise comparisons are the core of the approach for ranking refactoring patterns described in Chapter

4. The metrics are also used in Chapter 6 and in Chapter 5 to define heuristic rules for the prioritisation of refactoring opportunities. The formal definitions described in Chapter 9 are also used to define impact functions for the *Pull Up Advice* refactoring pattern in Chapter 7. The ten sample applications were used in Chapter 8 in a case study to evaluate the approach for simplifying refactoring sequences. Last, Chapter 3 describes how the metrics are used throughout several refactoring activities.

10.6 An Approach to Evaluate the Effects of Refactoring on Software Quality

This thesis proposes an explicit rationale to create impact functions for aspect-oriented refactoring patterns. Such impact functions are used to quantitatively predict the effects of refactoring on software quality. This is accomplished by the definitions of functions which compute the expected changes in the values of metrics when applying refactoring patterns.

The use of these functions to predict the impact on software quality can lead to better decisions while modifying existing software applications. The results show that the the developers can use the values computed by the impact functions to choose between different refactoring opportunities.

The approach is evaluated by defining (i) impact functions for an aspect-oriented refactoring pattern, named *Pull Up Advice*, for four metrics: lines of code, number of operations in module, crosscutting degree of an aspect and coupling on advice execution (Chapter 7), (ii) impact functions for six object-oriented refactoring patterns: *Copy Attribute*, *Copy Method*, *Delete Method*, *Move Method*, *New Subclass*, and *Extract Subclass* (Appendix E), and (iii) two additional impact functions for aspect-oriented software: *Extract Pointcut* and *Inline Inter-Type Declaration* (Appendix F). One case study was conducted to compute the values of impact functions for the *Pull Up Advice* in the Glassbox Inspector application, described in Chapter 9.

Impact functions are additional tools in the developer's tool kit to prioritise and evaluate refactoring opportunities, as they complement the use of the metric-based heuristic rules described in Chapter 6, by providing quantitative information about the effects that each refactoring application has in the software elements. As such, impact functions can help in the definition of pairwise comparisons between refactoring patterns (one of the key issues in ranking refactoring patterns - Chapter 4), in the prioritisation of refactoring opportunities (Chapters 5 and 6) and in evaluating refactoring sequences (Chapter 8). Chapter 3 describes the main functional requirements for tool support aiming at computing impact functions for refactoring patterns and metrics.

10.7 An Approach to Reduce the Number of Refactoring Sequences

During software development and evolution activities, the developers focus the refactoring efforts on choosing and applying refactoring patterns (or sequences of patterns) that are likely to improve the software quality. Considering the search for opportunities for applying refactoring sequences, the main problem is the number of possible sequences to be evaluated. This thesis proposes an approach to narrow the number of refactoring sequences by first creating the set of all possible sequences and then avoiding sequences of refactoring patterns that lead to the same results.

To evaluate the approach for reducing the search space for refactoring opportunities,

this thesis shows how deterministic finite automata (DFAs) representing the applicable refactoring sequences in existing software can be created and simplified. The approach was exemplified using five refactoring patterns dealing with the manipulation of methods. The initial DFA was simplified and its size was reduced in 62% (considering the total number of paths to be evaluated). Furthermore, the approach was applied and the number of possible sequences was computed for ten software applications before and after the simplification of sequences. In those applications, the reduction rate was confirmed, varying from 57 to 60% of the number of initial sequences. The case study shows that, considering sequences for method manipulation, the number of sequences can be significantly reduced.

An API was created to allow the developers to create the bindings between refactoring patterns and the symbols of a particular programming language grammar. This API also provides mechanisms to create the combination of all the sequences between a set of selected refactoring patterns for a specified level. Future work is needed to allow the creation and application of simplification rules to the initial sequences.

This contribution is directly related to the identification and prioritisation of refactoring opportunities (Chapters 5 and 6) as the techniques proposed in those chapters can also be applied to refactoring sequences. The metrics described in Chapter 9 were used to compute the sequences in the sample applications (both before and after the simplification). The search for refactoring sequences can be performed in accordance to the set of activities described in Chapter 3.

10.8 Future Work

There are several opportunities for research in the area covered by this thesis. The following are possible continuations for parts of this work:

- **Additional reduction of the search space.** Additional techniques can be used to further reduce the scope of refactoring, including additional methods for selecting the refactoring patterns to be included in the search, the modules to be evaluated and the optimal parameters for each refactoring pattern. Future work on this area can focus on answering the question of which are the optimal sequences according to their improvements on the expected quality attributes of a software application, and on further techniques to reduce the search space.

- **Evaluate the effects of specific metrics and refactoring patterns.** Further investigation can be carried out to evaluate the applicability of impact functions for assessing the effects of refactoring in design models. Future work can also focus on applying the same approach to other refactoring patterns for aspect-oriented software and object-oriented software, and for other metrics

- **Improving tool support for the search of refactoring opportunities.** The provided implementation can be extended to support integration with IDEs and modelling environments. This integration includes the development of user interfaces, search options, and ordering methods for refactoring opportunities.

- **Applying the proposed activities in early stages of software development.** The use of heuristic rules to search for refactoring opportunities and the use of impact functions to predict the changes in software metrics can lead to better decisions when modelling software. The use of both heuristic rules and impact functions in

early stages of a software development process might be advantageous. Both design and analysis models can be evaluated using the proposed rationale.

- **Validation in large scale projects.** Another interesting future work is the application of the proposed activities in large scale projects, in order to evaluate how the proposed approaches and techniques work together in a practical setting. Such validation is easier to conduct in the context of cooperation projects and using extensive tool support.

REFERENCES

ALWIS, B. D. et al. Coding Issues in AspectJ. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, OOPSLA, 2., 2000, Washington, USA. **Proceedings...** New York: ACM Press, 2000.

BALAZINSKA, M. et al. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In: WORKING CONFERENCE ON REVERSE ENGINEERING, WCRE, 7., 2000, Washington, USA. **Proceedings...** Los Alamitos: IEEE Press, 2000. p.98 – 107.

BALDAN, P. et al. **Handbook of Graph Grammars and Computing by Graph Transformations**: Concurrency, Parallelism, and Distribution. New Jersey, USA: World Scientific Publishing Company, 1999. v.3, p.107–188.

BARTSCH, M.; HARRISON, R. An Evaluation of Coupling Measures for AspectJ. In: WORKSHOP ON LINKING ASPECT TECHNOLOGY AND EVOLUTION, LATE, 2006, Bonn, Germany. **Proceedings...** New York: ACM Press, 2006.

BARTSCH, M.; HARRISON, R. An Exploratory Study of the Effect of Aspect-Oriented Programming on Maintainability. **Software Quality Journal**, Hingham, USA, v.16, n.1, p.23–44, Mar. 2008.

BASILI, V. R. **Software Modeling and Measurement – The Goal/Question/Metric Paradigm**. Maryland, USA: University of Maryland, 1992. (Technical Report, CS-TR-2956).

BAXTER, G. et al. Understanding the Shape of Java Software. **SIGPLAN Notices**, New York, NY, USA, v.41, n.10, p.397–412, Oct. 2006.

BECK, K. **Extreme Programming Explained**: Embrace Change. Boston, USA: Addison-Wesley, 1999.

BERG, K.; CONEJERO, J.; CHITCHYAN, R. **AOSD Ontology 1.0**. [S.l.]: AOSD-Europe, 2005. (Technical Report, AOSD-Europe-UT-01).

BERTOA, M.; VALLECILLO, A. Quality Attributes for COTS Components. In: WORKSHOP ON QUANTITATIVE APPROACHES IN OBJECT-ORIENTED SOFTWARE ENGINEERING, QAOOSE, 6., 2002, Malaga, Spain. **Proceedings...** [S.l.: s.n.], 2002.

BEVILACQUA, M.; BRAGLIA, M. The Analytic Hierarchy Process Applied to Maintenance Strategy Selection. **Reliability Engineering & System Safety**, [S.l.], v.70, n.1, p.71 – 83, 2000.

BOEHM, B. W.; BROWN, J. R.; LIPOW, M. Quantitative Evaluation of Software Quality. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 2., 1976, San Francisco, USA. **Proceedings...** Los Alamitos: IEEE Press, 1976. p.592–605.

BOEHM, B. W.; IN, H. Identifying Quality-Requirement Conflicts. **IEEE Software**, Los Alamitos, v.13, n.2, p.25–35, 1996.

BOEHM, B. W.; SULLIVAN, K. J. Software Economics: A Roadmap. In: ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 24., 2000, Limerick, Ireland. **Proceedings...** New York: ACM Press, 2000. p.319–343.

BOGER, M.; STURM, T. Refactoring Browser for UML. In: INTERNATIONAL CONFERENCE NET.OBJECTDAYS CONFERENCE, 3., 2002, Erfurt, Germany. **Proceedings...** [S.l.: s.n.], 2002. p.366–377.

BOIS, B. D. **A Study of Quality Improvements by Refactoring**. 2006. PhD Thesis – Universiteit Antwerpen, Belgium.

BOIS, B. D.; MENS, T. Describing the Impact of Refactorings on Internal Program Quality. In: INTERNATIONAL WORKSHOP ON EVOLUTION OF LARGE-SCALE INDUSTRIAL SOFTWARE APPLICATIONS, ELISA, 2003, Amsterdam, The Netherlands. **Proceedings...** [S.l.: s.n.], 2003.

BRIAND, L.; MORASCA, S.; BASILI, V. Property-Based Software Engineering Measurement. **IEEE Transactions on Software Engineering**, Los Alamitos, v.22, n.1, p.68–86, Jan. 1996.

BRITO, I. S. et al. Handling Conflicts in Aspectual Requirements Compositions. In: RASHID, A.; AKSIT, M. (Ed.). **Transactions on Aspect Oriented Software Development (TAOSD)**. Berlin: Springer-Verlag, 2007. p.144–166. (Lecture Notes in Computer Science, v.4620).

CACHO, N. et al. Composing Design Patterns – A Scalability Study of Aspect-Oriented Programming. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 5., 2006, Bonn, Germany. **Proceedings...** New York: ACM Press, 2006. p.109–121.

CASTOR FILHO, F. et al. Exceptions and Aspects: the devil is in the details. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, FSE, 14., 2006, Portland, USA. **Proceedings...** New York: ACM Press, 2006. p.152–162.

CASTOR FILHO, F.; GARCIA, A.; RUBIRA, C. A Quantitative Study on the Aspectization of Exception Handling. In: WORKSHOP ON EXCEPTION HANDLING IN OBJECT-ORIENTED SYSTEMS, 2005. **Proceedings...** [S.l.: s.n.], 2005.

CAVANO, J.; MCCALL, J. A Framework for the Measurement of Software Quality. In: SOFTWARE QUALITY ASSURANCE WORKSHOP ON FUNCTIONAL AND PERFORMANCE ISSUES, 1978. **Proceedings...** New York: ACM Press, 1978. p.133–139.

CECCATO, M.; TONELLA, P. Measuring the Effects of Software Aspectization. In: WORKSHOP ON ASPECT REVERSE ENGINEERING, WARE, 2004, Delft, The Netherlands. **Proceedings...** Los Alamitos: IEEE Press, 2004.

CHAVEZ, C. V. F. G.; LUCENA, C. J. P. de. Guidelines for Aspect-Oriented Design. In: BRAZILIAN WORKSHOP ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, WASP, 1., 2004, Brasília, Brazil. **Proceedings...** [S.l.: s.n.], 2004.

CHEN, S.-J. G.; LIN, L. Decomposition of inderdependent task group for concurrent engineering. **Computers & Industrial Engineering**, [S.l.], v.44, n.3, p.435–459, 2003.

CHENG, E. W. L.; LI, H. Construction Partnering Process and Associated Critical Success Factors: quantitative investigation. **Journal of Management in Engineering**, [S.l.], v.18, n.4, p.194–202, 2002.

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object Oriented Design. **IEEE Transactions on Software Engineering**, Los Alamitos, v.20, n.6, p.476–493, 1994.

COCKBURN, A.; HIGHSMITH, J. Agile Software Development: The People Factor. **Computer**, Los Alamitos, v.34, n.11, p.131–133, Nov. 2001.

COPLIEN, J.; HARRISON, N. **Organizational patterns of agile software development**. [S.l.]: Pearson Prentice Hall, 2005.

CORNELIO, M. **Refactorings as Formal Refinements**. 2004. PhD Thesis – Universidade Federal de Pernambuco, Brazil.

DEURSEN, A. van; MARIN, M.; MOONEN, L. AJHotDraw: A Showcase for Refactoring to Aspects. In: LINKING ASPECT TECHNOLOGY AND EVOLUTION, LATE, 1., 2005, Chicago, USA. **Proceedings...** New York: ACM Press, 2005.

DOUENCE, R.; FRADET, P.; SUDHOLT, M. A Framework for the Detection and Resolution of Aspect Interactions. In: ACM SIGPLAN/SIGSOFT CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, GPCE, 2002, London, UK. **Proceedings...** Berlin: Springer-Verlag, 2002. p.173–188.

DUCASSE, S.; RIEGER, M.; DEMEYER, S. A Language Independent Approach for Detecting Duplicated Code. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, ICSM, 1999, Oxford, England. **Proceedings...** Los Alamitos: IEEE Press, 1999. p.109–119.

ELRAD, T.; FILMAN, R.; BADER, A. Aspect-Oriented Programming. **Communications of the ACM**, New York, v.44, n.10, p.29–32, 2001.

ELSSAMADISY, A.; SCHALLIOL, G. Recognizing and Responding to Bad Smells in Extreme Programming. In: ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 26., 2002, Orlando, USA. **Proceedings...** New York: ACM Press, 2002. p.617–622.

EMDEN, E. v. Java Quality Assurance by Detecting Code Smells. In: WORKING CONFERENCE ON REVERSE ENGINEERING, WCRE, 9., 2002, Richmond, USA. **Proceedings...** Los Alamitos: IEEE Press, 2002. p.97–108.

FENTON, N. E.; PFLEEGER, S. L. **Software Metrics**: A Rigorous and Practical Approach. [S.l.]: PWS Publishing Company, 1997.

FILMAN, R.; FRIEDMAN, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, AOP, 2., 2000, Minneapolis, USA. **Proceedings...** New York: ACM Press, 2000.

FOWLER, M. et al. **Refactoring**: Improving the Design of Existing Code. [S.l.]: Addison-Wesley, 1999. Object Technologies Series.

GAMMA, E. et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. [S.l.]: Addison-Wesley, 1995. Addison-Wesley Professional Computing Series.

GARCIA, A. F. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. In: RASHID, A.; AKSIT, M. (Ed.). **Transactions on Aspect-Oriented Software Development II**. Berlin: Springer-Verlag, 2006. p.36–74, 2006. (Lecture Notes in Computer Science, v.4242).

GARCIA, A. F. et al. Modularizing Design Patterns with Aspects: A quantitative study. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 4., 2005, Chicago, USA. **Proceedings...** New York: ACM Press, 2005. p.3–14.

GARCIA, V. C. et al. Manipulating Crosscutting Concerns. In: LATIN AMERICAN CONFERENCE ON PATTERNS LANGUAGES OF PROGRAMMING, SUGARLOAF-PLOP, 4., 2004, Porto das Dunas, Brazil. **Proceedings...** [S.l.: s.n.], 2004.

GELDERMANN, J.; SPENGLER, T.; RENTZ, O. Fuzzy Outranking for environmental assessment. Case Study: Iron and Steel Making Industry. **Fuzzy Sets and Systems**, [S.l.], v.115, n.1, p.45 – 65, 2000.

GREENWOOD, P.; BLAIR, L. A Framework for Policy-Driven Auto-Adaptive Systems Using Dynamic Framed Aspects. In: RASHID, A.; AKSIT, M. (Ed.). **Transactions on Aspect-Oriented Software Development II**. Berlin: Springer-Verlag, 2006. p.30 – 65, 2006. (Lecture Notes in Computer Science, v.4242).

HALL, P. A. V.; DOWLING, G. R. Approximate String Matching. **ACM Computing Surveys**, New York, v.12, n.4, p.381–402, 1980.

HANENBERG, S.; OBERSCHULTE, C.; UNLAND, R. Refactoring of Aspect-Oriented Software. In: INTERNATIONAL CONFERENCE NET.OBJECTDAYS CONFERENCE, 4., 2003, Erfurt, Germany. **Proceedings...** [S.l.: s.n.], 2003.

HANENBERG, S.; UNLAND, R. Using and Reusing Aspects in AspectJ. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, AOP, 3., 2001, Tampa Bay, USA. **Proceedings...**, New York: ACM Press, 2001.

HANNEMANN, J.; KICZALES, G. Design Patterns Implementation in Java and AspectJ. In: OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA, 17., 2002, Seattle, USA. **Proceedings...** New York: ACM Press, 2002. p.161–173.

HECHT, M. V.; PIVETA, E. K.; PIMENTA, M. S.; PRICE, R. T. Aspect-Oriented Code Generation. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES, 21., 2006, Florianópolis, Brazil. **Proceedings...** Porto Alegre: SBC, 2006.

HECKEL, R.; KUSTER, J.; TAENTZER, G. Confluence of Typed Attributed Graph Transformation Systems. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, ICGT, 1., 2002, Barcelona, Spain. **Graph Transformation:** proceedings. Berlin: Springer-Verlag, 2002. p.161–176. (Lecture Notes in Computer Science, v.2505).

HILSDALE, E.; KICZALES, G. Aspect-Oriented Programming with AspectJ. In: OBJECT ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, 16., 2001, Tampa Bay, USA. **Tutorial**. New York: ACM, 2001.

ISO. **ISO/IEC 9126-1**: Software Engineering – Product Quality – Part 1 – Quality Model. [S.l.], 2001.

IWAMOTO, M.; ZHAO, J. Refactoring Aspect-Oriented Programs. In: AOSD MODELING WITH UML WORKSHOP, AOM, 5., 2003, San Francisco, USA. **Proceedings...** [S.l.: s.n.], 2003.

JARO, M. Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. **Journal of the American Statistical Association**, [S.l.], v.84, n.406, p.414–420, 1989.

JOERIN, F.; MUSY, A. Land Management with GIS and Multicriteria Analysis. **International Transactions in Operational Research**, [S.l.], v.7, n.1, p.67–78, 2000.

JOHNSON, R. E. Documenting Frameworks Using Patterns. In: OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA, 17., 2002, Vancouver, Canada. **Proceedings...** New York: ACM Press, 2002.

JUNG, H.-W. Optimizing Value and Cost in Requirements Analysis. **IEEE Software**, Los Alamitos, v.15, n.4, p.74–78, 1998.

JUNG, H.-W.; CHOI, B. Optimization Models for Quality and Cost of Modular Software Systems. **European Journal of Operational Research**, [S.l.], v.112, n.3, p.613–619, 1999.

KATAOKA, Y. et al. Automated Support for Program Refactoring Using Invariants. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, ICSM, 2001, Florence, Italy. **Proceedings...** Los Alamitos: IEEE Press, 2001. p.736–743.

KATAOKA, Y. et al. A Quantitative Evaluation of Maintainability Enhancement by Refactoring. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, ICSM, 18., 2002, Montreal, Canada. **Proceedings...** Los Alamitos: IEEE Press, 2002. p.576–585.

KAZMAN, R.; BASS, L. **Toward Deriving Software Architectures From Quality Attributes**. [S.l.]: Software Engineering Institute - Carnegie Mellon University, Pennsylvania - USA, 1994. (CMU/SEI-94-TR-010).

KERIEVSKY, J. **Refactoring to Patterns**. [S.l.]: Addison-Wesley, 2005. Addison-Wesley Signature Series.

KICZALES, G. et al. An Overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, ECOOP, 15., 2001, Budapest, Hungary. **Proceedings...** New York: ACM Press, 2001. p.327–353.

KICZALES, G. et al. Getting Started with AspectJ. **Communications of the ACM**, New York, v.44, n.10, p.59–65, 2001.

KICZALES, G. et al. Aspect-Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, ECOOP, 11., 1997, Jyvaskyla, Finland. **Proceedings...** Berlin: Springer-Verlag, 1997. p.220–242.

KOPPEN, C.; STORZER, M. PCDiff: Attacking the Fragile Pointcut Problem. In: EUROPEAN INTERACTIVE WORKSHOP ON ASPECTS IN SOFTWARE, EIWAS, 2004, Berlin, Germany. **Proceedings...** [S.l.]: Vrije Universiteit Brussel, 2004.

KRUTCHEN, P. **The Rational Unified Process**: An Introduction. [S.l.]: Addison Wesley, 2000.

LANZA, M.; DUCASSE, S. Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics. In: LANGAGES ET MODELES A OBJETS, LMO, 2002. **Proceedings...** [S.l.: s.n.], 2002.

LAVAZZA, L. Providing Automated Support for the GQM Measurement Process. **IEEE Software**, Los Alamitos, v.17, n.3, p.56–62, 2000.

LIU, H.; LI, G.; MA, Z.; SHAO, W. Scheduling of conflicting refactorings to promote quality improvement. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE, 22., 2007, Atlanta, USA. **Proceedings...** New York: ACM Press, 2007. p.489–492.

LOPES, C. V. **D – A Language Framework for Distributed Programming**. 1997. PhD Thesis – College of Computer Science, Northeastern University, USA.

MAHRENHOLZ, D.; SPINCZYK, O.; SCHRODER-PREIKSCHAT, W. Program Instrumentation for Debugging and Monitoring with AspectC++. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, ISORC, 5., 2002, Crystal City, USA. **Proceedings...** Los Alamitos: IEEE Press, 2002. p.249–256.

MARKOVIC, S.; BAAR, T. Refactoring OCL Annotated UML Class Diagrams. In: MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, MODELS, 8., 2005, Montego Bay, Jamaica. **Proceedings...** [S.l.: s.n.], 2005. p.280–294.

MCCALL, J.; RICHARDS, P.; WALTERS, G. **Factors in Software Quality**. Sunnyvale, USA: General Electric Ed., 1977. Technical Report.

MENS, T. et al. Refactoring: current research and future trends. **Electronic Notes in Theoretical Computer Science**, Amsterdam, The Netherlands, v.82, n.3, p.483–499, 2003.

MENS, T.; TAENTZER, G.; RUNGE, O. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. **Electronic Notes in Theoretical Computer Science**, Amsterdam, The Netherlands, v.127, n.3, p.113–128, 2005.

MENS, T.; TOURWE, T. A Survey of Software Refactoring. **IEEE Transactions on Software Engineering**, Los Alamitos, v.30, n.2, p.126–139, 2004.

MEZINI, M.; OSTERMANN, K. Conquering aspects with Caesar. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2003, Boston, USA. **Proceedings...** New York: ACM Press, 2003. p.90–99.

MONTEIRO, M. P.; FERNANDES, J. M. Object-to-Aspect Refactorings for Feature Extraction. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 2004, Lancaster, UK. **Proceedings...** New York: ACM Press, 2004.

MONTEIRO, M. P.; FERNANDES, J. M. Towards a Catalog of Aspect-Oriented Refactorings. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 4., 2005, Chicago, USA. **Proceedings...** New York: ACM Press, 2005. p.111–122.

MONTEIRO, M. P.; FERNANDES, J. M. The Search for Aspect-Oriented Refactorings Must Go On. In: LINKING ASPECT TECHNOLOGY AND EVOLUTION, LATE, 1., 2005, Chicago, USA. **Proceedings...** New York: ACM Press, 2005.

MONTEIRO, M. P.; FERNANDES, J. M. Towards a Catalogue of Refactorings and Code Smells for AspectJ. In: RASHID, A.; AKSIT, M. (Ed.). **Transactions on Aspect-Oriented Software Development II**. Berlin: Springer-Verlag, 2006. p.214–258, 2006. (Lecture Notes in Computer Science, v.4242).

MOREIRA, A. M. D.; ARAUJO, J.; BRITO, I. S. Crosscutting Quality Attributes for Requirements Engineering. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, SEKE, 14., 2002, Ischia, Italy. **Proceedings...** [S.l.: s.n.], 2002. p.167–174.

MOSER, R. et al. Does Refactoring Improve Reusability? In: ICSR, 2006. **Proceedings...** [S.l.: s.n.], 2006. p.287–297.

MYLOPOULOS, J.; CHUNG, L.; NIXON, B. A. Representing and Using Non-Functional Requirements: A Process-Oriented Approach. **IEEE Transactions on Software Engineering**, Los Alamitos, v.18, n.6, p.483–497, 1992.

MYSQL. **MySql ConnectorJ Home Page**. Disponível em: <http://www.mysql.com/products/connector/j/>. 2009. Acesso em: 22 jan. 2009.

NAVARRO, G. A Guided Tour to Approximate String Matching. **ACM Computing Surveys**, New York, v.33, n.1, p.31–88, 2001.

OFFUTT, J. Quality Attributes of Web Software Applications. **IEEE Software**, Los Alamitos, v.19, n.2, p.25–32, 2002.

OPDYKE, W. F. **Refactoring Object-Oriented Frameworks**. 1992. PhD Thesis – University of Illinois at Urbana Champaign, USA.

PIVETA, E. K.; ARAUJO, J.; MOREIRA, M. S. P. A. M. D.; ; GUERREIRO, P.; PRICE, R. T. Searching for Opportunities of Refactoring Sequences: Reducing the Search Space. In: IEEE COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 32., 2008, Turku, Finland. **Proceedings...** Los Alamitos: IEEE Press, 2008.

PIVETA, E. K.; HECHT, M.; PIMENTA, M. S.; PRICE, R. T. Detecting Bad Smells in AspectJ. **Journal of Universal Computer Science**, [S.l.], v.12, n.7, p.811–827, July 2006.

PIVETA, E. K.; HECHT, M. V.; MOREIRA, A. M. D.; PIMENTA, M. S.; ARAUJO, J.; GUERREIRO, P.; PRICE, R. T. Avoiding Bad Smells in Aspect-Oriented Software. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, SEKE, 19., 2007, Boston, USA. **Proceedings...** [S.l.: s.n.], 2007.

PIVETA, E. K.; HECHT, M. V.; PIMENTA, M. S.; PRICE, R. T. Bad Smells em Sistemas Orientados a Aspectos. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES, 20., 2005, Uberlândia, Brazil. **Proceedings...** Porto Alegre: SBC, 2005.

PIVETA, E. K.; HECHT, M. V.; PIMENTA, M. S.; PRICE, R. T. Detecting Bad Smells in AspectJ. In: BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, SBLP, 10., 2006, Itatiaia, Brazil. **Proceedings...** Porto Alegre: SBC, 2006.

PIVETA, E. K.; MOREIRA, A. M. D.; PIMENTA, M. S.; ARAUJO, J.; GUERREIRO, P.; PRICE, R. T. Ranking Refactoring Patterns with the Analytic Hierarchy Process. In: INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, ICEIS, 10., 2008, Barcelona, Spain. **Proceedings...** [S.l.: s.n.], 2008.

PIVETA, E. K.; PIMENTA, M. S.; ARAUJO, J.; MOREIRA, A. M. D.; ; GUERREIRO, P.; PRICE, R. T. Representing Refactoring Opportunities. In: ANUAL ACM SYMPOSIUM ON APPLIED COMPUTING, SAC, 24., 2009, Honolulu, EUA. **Proceedings...** New York: ACM Press, 2009.

RAMOS, R. A.; PIVETA, E. K.; CASTRO, J.; ARAUJO, J.; MOREIRA, A. M. D.; GUERREIRO, P.; PIMENTA, M. S.; PRICE, R. T. Improving the Quality of Requirements with Refactoring. In: BRAZILIAN SYMPOSIUM ON SOFTWARE QUALITY, SBQS, 6., 2007, Porto de Galinhas, Brazil. **Proceedings...** Porto Alegre: SBC, 2007.

RUI, K.; REN, S.; BUTLER, G. Refactoring Use Case Models: a case study. In: INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, ICEIS, 5., 2003, Angers, France. **Proceedings...** [S.l.: s.n.], 2003. p.239–244.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence**: A Modern Approach. 2nd ed. [S.l.]: Prentice Hall, 2002.

SAATY, T. L. How to Make a Decision: the analytic hierarchy process. **European Journal of Operational Research**, [S.l.], v.48, n.1, p.9 – 26, 1990.

SAATY, T. L. Decision-Making With the AHP: why is the principal eigenvector necessary? **European Journal of Operational Research**, [S.l.], v.145, n.1, p.85 – 91, 2003.

SALTON, G.; MCGILL, M. **Introduction to Modern Information Retrieval**. New York, USA: McGraw-Hill, 1986.

SANTANNA, C. et al. On the Reuse and Maintenance of AO Software: An assessment framework. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 17., 2003, Manaus, Brazil. **Proceedings...** Porto Alegre: SBC, 2003.

SCHWABER, K. Scrum Development Process. In: ACM CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA, 10., 1995, Austin, USA. **Proceedings...** New York: ACM Press, 1995. p.117–134.

SHIMAZAKI, H. **Recipes for Selecting the Bin Size of a Histogram**. 2006. PhD Thesis – Kyoto University, Japan.

SIMON, F.; STEINBRUCKNER, F.; LEWERENTZ, C. Metrics Based Refactoring. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, CSMR, 5., 2001, Lisbon, Portugal. **Proceedings...** [S.l.: s.n.], 2001. p.30–38.

SIPSER, M. **Introduction to the Theory of Computation**. [S.l.]: International Thomson Publishing, 1996.

SNEDECOR, G. et al. **Statistical Methods**. 8th ed. [S.l.]: Blackwell Publishing, 1989.

SOARES, S.; LAUREANO, E.; BORBA, P. Implementing distribution and persistence aspects with AspectJ. In: OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGE AND APPLICATIONS, OOPSLA, 17., 2002, Seattle, USA. **Proceedings...** New York: ACM Press, 2002. p.174–190.

SRIVISUT, K.; MUENCHAISRI, P. Defining and Detecting Bad Smells of Aspect-Oriented Software. In: IEEE INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 2007. **Proceedings...** Los Alamitos: IEEE Press, 2007. p.65–70.

STOCHMIALEK, M. **Aopmetrics - Project Home Page**. Disponível em: <http://aopmetrics.tigris.org/>. 2009. Acesso em: 22 jan. 2009.

SUNYE, G. et al. Refactoring UML Models. In: THE UNIFIED MODELING LANGUAGE INTERNATIONAL CONFERENCE, UML, 4., 2001, Toronto, Canada. **Proceedings...** [S.l.: s.n.], 2001. p.134–148.

TEAM, C. **Capability Maturity Model® Integration (CMMI SM), Version 1.1**. [S.l.]: Pitsburg, Software Engineering Institute, 2001.

TONELLA, P.; CECCATO, M. Aspect Mining through the Formal Concept Analysis of Execution Traces. In: WORKING CONFERENCE ON REVERSE ENGINEERING, WCRE, 11., 2004. **Proceedings...** [S.l.: s.n.], 2004.

TOURWE, T.; MENS, T. Identifying Refactoring Opportunities Using Logic Meta Programming. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, CSMR, 7., 2003, Benevento, Italy. **Proceedings...** [S.l.: s.n.], 2003. p.91–100.

TSANG, S. L.; CLARKE, S.; BANIASSAD, E. L. A. An Evaluation of AOP for Java-Based Real-Time Systems Development. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, ISORC, 7., 2004, Vienna, Austria. **Proceedings...** [S.l.: s.n.], 2004. p.291–300.

VANHAUTE, B.; WIN, B.; DECKER, B. Building Frameworks in AspectJ. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, AOP, 3., 2001, Budapest, Hungary. **Proceedings...** [S.l.: s.n.], 2001.

WEYUKER, E. Evaluating Software Complexity Measures. **IEEE Transactions Software Engineering**, Los Alamitos, v.14, n.9, p.1357–1365, 1988.

XU, J. et al. Use Case Refactoring: a tool and a case study. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, APSEC, 11., 2004, Busan, Korea. **Proceedings...** [S.l.: s.n.], 2004. p.484–491.

YU, W.; LI, J.; BUTLER, G. Refactoring Use Case Models on Episodes. In: IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE, 19., 2004, Linz, Austria. **Proceedings...** Los Alamitos: IEEE Press, 2004. p.328–335.

ZAKARIA, A.; HOSNY, H. Metrics for Aspect-Oriented Software Design. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELLING, 3., 2003, Boston, USA. **Proceedings...** New York: ACM Press, 2003.

ZHANG, C. et al. Aspect Refactoring Verifier. In: LINKING ASPECT TECHNOLOGY AND EVOLUTION, LATE, 1., 2005, Chicago, USA. **Proceedings...** [S.l.: s.n.], 2005.

ZHANG, J.; LIN, Y.; GRAY, J. Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine. In: MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, MODELS, 8., 2005, Toronto, Canada. **Proceedings...** [S.l.: s.n.], 2005. p.199–217.

ZHAO, J. **Towards a Metrics Suite for Aspect-Oriented Software**. [S.l.]: Information Processing Society of Japan (IPSJ), 2002. (SE-136-25).

ZHAO, J. Measuring Coupling in Aspect-Oriented Systems. In: INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, METRICS, 10., 2004, Chicago, USA. **Proceedings...** [S.l.: s.n.], 2004.

ZHAO, J.; XU, B. Measuring Aspect Cohesion. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, FASE, 7., 2004. **Proceedings...** [S.l.: s.n.], 2004.

GLOSSARY

Analytical Hierarchy Process

The Analytical Hierarchy Process (AHP) (SAATY, 1990) is a mathematical decision making technique for evaluating a set of different alternative solutions of a given problem. It focuses on finding an optimal solution using qualitative and quantitative decision analysis. Pág. 33

Activity

An activity is a major task expressed in a process that must take place in order to fulfil an operation contract. Pág. 38

Advice

An advice is a method-like construction, which specifies a certain behaviour triggered by an associated pointcut. Pág. 29

Artefact

An artefact is a document needed or generated while accomplishing the tasks described in each activity. Pág. 38

Aspect

An aspect is an abstraction mechanism designed specifically to deal with crosscutting concerns. It is comprised by fields, methods, join-points, pointcuts, advices and inter-type declarations. Pág. 29

Catalogue of Quality Attributes

A catalogue of quality attributes is an organised collection that provide details of how each quality attribute can affect the quality of software, of related quality attributes, and of possible trade-offs between conflicting quality attributes. Pág. 18

Discipline

A discipline is an organised set of inter-related activities which encapsulates a core concern of the process. Pág. 38

Goal/Question/Metric (GQM)

GQM is an approach to software metrics that defines a measurement model, comprised of three levels: conceptual level (goals), operational level (questions) and quantitative levels (metrics). Pág. 92

Heuristic Rule

In this thesis, a heuristic rule is a function created to quantitatively evaluate a set of quality attributes of a given software element. Pág. 92

Inter-Type Declaration

Inter-type declarations are constructions of programming languages used to introduce state or behaviour to an existing class, aspect or interface. Pág. 29

Join Point

A join point is a well-defined point in the execution flow of a program. Examples of join points are: method and constructor calls and execution, field access and initializations. Pág. 28

Model Element

A model element can be any element in a model, such as a class, an attribute, a relationship, or a use case. Pág. 53

Pointcut

Pointcuts are constructions of programming languages that group join points by the definition of a predicate that, whenever satisfied, causes the actions associated to it to be executed. Pág. 29

Quality Attribute

In software engineering, a *quality attribute* is a requirement which specifies criteria that can be used to judge the operation of a system (ISO, 2001). Other terms for quality attributes are non-functional requirements, constraints, quality goals and quality of service requirements. Pág. 18

Quality Model

Quality models group quality attributes together, organising hierarchically and refining them to express the expected quality of a specific software system or of a specific domain. Pág. 19

Ranking Method

A ranking method is comprised by a set of steps to rank refactoring patterns according to a quality model. Pág. 68

Ranking of Refactoring Patterns

A ranking of refactoring patterns is an ordered list of refactoring patterns according to a set of selected quality attributes. The closer a refactoring pattern is from the top of the list, the higher is the impact of the refactoring pattern on the selected quality attributes. (see also *Refactoring Pattern* and *Quality Attribute*). Pág. 67

Refactoring

Refactoring (OPDYKE, 1992; FOWLER et al., 1999) is the process of improving the design of software systems without changing their externally observable behaviour. Pág. 17

Refactoring Opportunity

A refactoring opportunity is defined as the association between a software element, a shortcoming, and a refactoring pattern (see also *Software Element* and *Refactoring Pattern*). Pág. 17

Refactoring Pattern

A refactoring pattern is a behavioural preserving transformation comprised by a name, a set of parameters, a motivation, a set of steps for applying it, the definition of pre and postconditions, and an example. Pág. 17

Refactoring to Pattern

A refactoring to pattern is a refactoring pattern that, when applied, changes the design of a software element to use a specific design pattern. (see also *Refactoring Pattern* and *Software Element*). Pág. 24

Shortcoming

In the context of refactoring, a shortcoming is a deficiency, inadequacy or incompleteness that a software element can have. Shortcomings typically affect negatively the quality attributes of a software system. Pág. 20

Software Development Process

A software development process is usually composed by a set of interrelated activities, a set of roles and their association with the process activities, artefacts needed and artefacts created or modified by the activities. Pág. 38

Software Element

A software element is any piece of software ranging from source code to analysis and design models. For example, in the context of object-oriented source code, software elements can be classes, methods, attributes. In use case diagrams, they are the actors, use cases, relationships etc. Pág. 19

Test Case

A test case is a set of conditions or variables under which it is possible to determine if a requirement is partially or fully satisfied. Pág. 40

APPENDIX A PAPERS

In the context of this thesis, nine papers were accepted: one paper in a Qualis B International Journal (1 PI-B), three papers in Qualis A International Conferences (3 CI-A), one paper in a Qualis B International Conference (1 CI-B), three papers in Qualis A National Conferences (3 CN-A), and one paper in a Qualis B National Conference (1 CN-B). These papers are described in more details as follows.

In 2008, three papers were accepted. The first one, accepted in the ICEIS Conference (PIVETA et al., 2008), shows how a set of refactoring patterns can be ranked to improve a set of quality attributes. This paper describes a quantitative and qualitative method for the ranking generation according to a quality model. The second one, accepted in the IEEE COMPSAC Conference (PIVETA et al., 2008), deals with the reduction of the search space when searching for refactoring opportunities in the context of refactoring sequences. Non-deterministic finite automata are used to simulate the refactoring sequences together with a set of rules for path simplification. The last one, accepted in the ACM SAC'09 Conference (PIVETA et al., 2009), describes how to represent refactoring opportunities in a language independent way, how to express in which conditions it is advantageous to apply refactoring patterns, and which refactoring patterns are associated with each condition.

In 2007, two papers were accepted. The first one, accepted in the SEKE Conference (PIVETA et al., 2007), describes a set of guidelines to avoid the occurrence of shortcomings in aspect-oriented software. The second paper, accepted in the SBQS Conference (RAMOS et al., 2007) deals with a set of refactoring patterns to requirements documents and how these refactoring patterns can improve metric values which are used to evaluate those documents.

In 2006, three papers were accepted. The first one, accepted at SBLP Conference (PIVETA et al., 2006b) describes algorithms to detect shortcomings in aspect-oriented code. This paper was later extended to a journal version (PIVETA et al., 2006a). The third paper, accepted in the SBES Conference (HECHT et al., 2006), describes how to generate aspect-oriented code using a set of design models described in Theme/UML, an aspect-oriented modelling language.

In 2005, one paper was accepted, in the SBES Conference (PIVETA et al., 2005). This paper describes typical shortcomings in aspect-oriented software. Each shortcoming is described by a name, a context, an example and a set of refactoring patterns which can be used to minimize its effects.

In summary, the following papers were accepted and published in peer reviewed venues:

ACM SAC'09 - 24th Annual ACM Symposium on Applied Computing
Representing Refactoring Opportunities
E. Piveta, M. Pimenta, J. Araujo, A. Moreira, P. Guerreiro, R. Tom Price
Qualis: CI-A

IEEE COMPSAC'08 - 32nd Annual IEEE International Computer Software and Applications Conference
Searching for Opportunities of Refactoring Sequences: Reducing the Search Space
E. Piveta, J. Araujo, M. Pimenta, A. Moreira, P. Guerreiro, R. Tom Price
Qualis: CI-A

ICEIS'08 - 10th International Conference on Enterprise Information Systems
Ranking Refactoring Patterns with the Analytic Hierarchy Process
E. Piveta, A. Moreira, M. Pimenta, J. Araujo, P. Guerreiro, R. Tom Price
Qualis: CI-A

SEKE'07 - 19th International Conference on Software Engineering and Knowledge Engineering
Avoiding Bad Smells in AO Software
E. Piveta, M. Hecht, A. Moreira, M. Pimenta, J. Araujo, P. Guerreiro, R. Tom Price
Qualis: CI-B

SBQS'07 - 6th Brazilian Symposium on Software Quality
Improving the Quality of Requirements with Refactoring
R. Ramos, E. Piveta, J. Castro, J. Araujo, A. Moreira, P. Guerreiro, M. Pimenta, R. Tom Price
Qualis: CN-B

JUCS'06 - Journal of Universal Computer Science
Detecting Bad Smells in AspectJ
E. Piveta, M. Hecht, M. Pimenta, R. Tom Price
Qualis: PI-B

SBLP'06 - 10th Brazilian Symposium on Programming Languages
Detecting Bad Smells in AspectJ
E. Piveta, M. Hecht, M. Pimenta, R. Tom Price
Qualis: CN-A

SBES'06 - 20th Brazilian Symposium on Software Engineering
Aspect-Oriented Code Generation
M. Hecht, E. Piveta, M. Pimenta, R. Tom Price
Qualis: CN-A

SBES'05 - 19th Brazilian Symposium on Software Engineering
Bad Smells em Sistemas Orientados a Aspectos
E. Piveta, M. Hecht, M. Pimenta, R. Tom Price
Qualis: CN-A

APPENDIX B GUIDELINES TO AVOID SHORTCOMINGS IN ASPECT-ORIENTED SOFTWARE

This appendix defines guidelines to avoid certain shortcomings which can occur in aspect-oriented software applications. Each guideline is followed by a brief motivation and by an example chosen from a well-known set of aspect-oriented implementations of design patterns (HANNEMANN; KICZALES, 2002) and other cases from Alwis et al. (ALWIS et al., 2000), and Hilsdale and Kickzales (HILSDALE; KICZALES, 2001). This appendix shows both constructions using the guidelines and constructions that do not follow the guidelines.

It is organised as follows. Sections B.1 to B.5 describe each of the proposed guidelines. Section B.6 provides a discussion of the benefits of using the defined guidelines. Section B.7 concludes the appendix.

B.1 Use Abstract Aspects

Guideline. Design towards abstract aspects, whose behaviour is defined by its advices, and its relationship with classes or other aspects is accomplished by specialization or association.

Motivation. The use of abstract aspects can help in developing more reusable aspects, by postponing implementation decisions and leaving the definition of concrete pointcut definitions to the sub-aspects. Furthermore, the behaviour defined in abstract aspects can be reused to different target applications. Each application can create sub-aspects capturing the specific points that activate the aspect behaviour.

Example. In the *Observer* pattern (HANNEMANN; KICZALES, 2002), there is a *ScreenObserver* aspect that extends the reusable abstract aspect *ObserverProtocol* (line 1) and defines that both roles (*Subject* and *Observer*) will be played by the *Screen* class (lines 2 and 3). It also defines when the subject state changes (line 4) and what should be done to update the observers (lines 5-7). This example complies with this guideline, defining an abstract aspect implementing the logic for the *Observer* pattern and leaving for the sub-aspects to bind the *Subject* and *Observer* roles and the changes in the *Subject* with the classes that play these roles.

```
1  public aspect ScreenObserver extends ObserverProtocol{
2    declare parents: Screen implements Subject;
3    declare parents: Screen implements Observer;
4    pointcut subjectChange(Subject sub): call(void Screen.
       display(String)) && target(sub);
5    void updateObserver(Subject sub, Observer obs) {
```

```
6       ((Screen)obs).display("Updated");
7     }
8   }
```

In the *Decorator* pattern (HANNEMANN; KICZALES, 2002), the *BracketDecorator* (line 1) and the *StarDecorator* (line 5) aspects have a duplicated pointcut named *printCall* (lines 2 and 7). In the next example, the implementation does not follow the *use abstract aspects* guideline.

```
1  public aspect BracketDecorator {
2    protected pointcut printCall(String s):
3      call(public void ConcreteOutput.print(String)) && args(s);
4  }
5  public aspect StarDecorator {
6    declare precedence: StarDecorator, BracketDecorator;
7    protected pointcut printCall(String s):
8      call(public void ConcreteOutput.print(String)) && args(s);
9  }
```

This implementation can be improved by introducing a super-aspect containing the common pointcut, eliminating the duplication.

B.2 Use Named Pointcuts

Guideline. Use named pointcuts to provide hot spots for extension and to use the terms of the problem domain in hand.

Motivation. The definition of named pointcuts allows the reuse of the predicate associated with this pointcut. Furthermore, a new term is added to the vocabulary regarding the concern being encapsulated by the aspect. The use of names for pointcuts provides a terminology for the development of the system. It also allows that pointcuts are subjected to future refinement, by defining concrete pointcuts on sub-aspects.

Example. In the *Factory Method* design pattern (HANNEMANN; KICZALES, 2002) there is an aspect that changes the behaviour of a *Factory Method*, using an around advice. With this approach it is possible to have factories creating different products, depending on the aspects woven into the project. In the *AlternateLabelCreatorImplementation* (line 1), the developer defines a pointcut named *labelCreation* (line 2), used in the around advice. If a before or an after advice is needed, the developer can reuse the definition. This definition serves also to define composition of pointcut predicates, and can be used by other pointcuts in the aspect. Other examples of the application of this guideline can be seen in Kiczales et al. (KICZALES et al., 2001b).

```
1  public aspect AlternateLabelCreatorImplementation {
2    pointcut labelCreation(): execution(JComponent LabelCreator
       .createComponent());
3    JComponent around(): labelCreation(){
4      JLabel label = (JLabel) proceed();
5      label.setText("... alternate JLabel");
6      return label;
7    }
8  }
```

In the *Builder* design pattern (HANNEMANN; KICZALES, 2002), this guideline is not followed in an aspect named *CreatorImplementation*, as shown below. The pointcut predicate is defined directly in the declare construction (line 2) and therefore cannot be reused in other pointcuts.

```
1 public aspect CreatorImplementation {
2    declare error: (set(public String Creator+.representation)
3       || get(public String Creator+.representation)) && !(
          within(Creator+)
4       || within(CreatorImplementation)): "variable result is
          aspect protected...";
5 }
```

Other examples of unnamed pointcuts can be seen in the *QueueStateAspect*, that implements the state transitions for the *State* pattern, in the *SortingStrategy* aspect, implementing part of the *Strategy* pattern and in the *SingletonProtocol* aspect, which defines the general behaviour of the *Singleton* pattern (HANNEMANN; KICZALES, 2002).

B.3 Use Semantic Based Pointcuts

Guideline. Avoid relying only on names of methods and classes for pointcut composition. Instead, use annotation or inheritance mechanisms, to associate semantics to class members. You can use inheritance or interface implementation references in the pointcuts to provide clearly defined semantics.

Motivation. Sometimes, a naming convention is adopted during the development of a system. However, these conventions are not always obeyed, or they are inadequate when dealing with the representation of join points collections to be affected by an aspect. Naming mechanisms increase the coupling between the base system and aspects, are not checked and are not guaranteed to be followed (KOPPEN; STORZER, 2004).

Example. You can define semantic based pointcuts using inheritance, for example, as used in the *Singleton* design pattern (HANNEMANN; KICZALES, 2002). The *SingletonInstance* aspect defines a pointcut named *protectionExclusions* (line 3) specifying a predicate that is true every time an instance of *PrinterSubclass* (line 4) or one of its sub-classes is created. Another possibility is to associate predicates to interfaces, using the advantages of this construct.

```
1 public aspect SingletonInstance extends SingletonProtocol {
2    declare parents: Printer implements Singleton;
3    protected pointcut protectionExclusions():
4    call((PrinterSubclass+).new(..));
5 }
```

In the *Proxy* design pattern example (HANNEMANN; KICZALES, 2002), the *unsafeRequest* method (line 6) is directly mentioned in the *requests* pointcut predicate (line 2). As this example was created before the support for annotations in *AspectJ 5* it did not rely in the constructions regarding annotations. Using these constructs, the developer can define an annotation to define a security status of a method (@*safety*, for example) in order to be available to the *requests* pointcut.

```
1 public aspect RequestBlocking extends ProxyProtocol {
2    protected pointcut requests():
```

```
3         call(* OutputImplementation.unsafeRequest(..));
4   }
5   public class OutputImplementation {
6       public void unsafeRequest(String s) {
7         System.out.println("[OutputImplementation.unsafeRequest()
              ]: "+s);
8       }
9   }
```

B.4 Favour Pointcut Composition

Guideline. Every time a pointcut definition contains join points without a strong semantic relationship, favour specifying pointcuts as the combination of two or more distinct pointcuts, one for each well-defined set of join points.

Motivation. Sometimes, when defining a pointcut, the developer puts together a set of heterogeneous join points in the same predicate. The developer should focus on grouping related join points in separated pointcuts and composing these pointcuts in order to achieve the desired combination.

Example. Alwis et al. (ALWIS et al., 2000) employ an example that illustrates this principle, by separating the pointcut definition in different sets. Initially, there was a pointcut named *lowLevelDataOperations* (line 1) containing several method calls in its predicate. Operations related to an *ASCII* channel, to a binary channel, and to a list of commands are specified in a single pointcut definition, as seen below.

```
1  pointcut lowLevelDataOperations():
2        (target(AsciiDataChannel) && (call(String readLine(..))
             || call(void writeLine(..))))
3     || (target(BinaryDataChannel) && (call(long read(..)) ||
       call(void write(..))))
4     || (target(ListCommand) && call(void writeFileInfo(..))) ;
```

The separation of these definitions improves the readability of the pointcut and allows the developer to reuse the new pointcuts. It is also easier to evolve the aspect, as each set of related method calls are defined in a separated pointcut. Consider the example below. The *lowLevelDataOperations* (line 1) is now composed by several pointcut definitions: *asciiDataOps*, *binaryDataOps* and *listCommandDataOps* (lines 2-4). Below, there is the code complying with this guideline.

```
1  pointcut lowLevelDataOperations() : asciiDataOps() ||
     binaryDataOps() || listCommandDataOps();
2  pointcut asciiDataOps(): ... ;
```

Pointcuts related to a single class (such as *asciiDataChannelDataOps*, line 1) can be moved to this class. This would simplify the maintenance of both the class and the aspect. The disadvantage of this approach is the difficulty in the comprehension of the aspect just by reading its definition; it becomes necessary to find pointcuts in other classes of the system.

In the Facade pattern (HANNEMANN; KICZALES, 2002) implementation, the *FacadePolicyEnforcement* aspect defines a declare warning construction that uses a composite pointcut to describe that a warning should be raised every time a encapsulated method

is called outside the facade (line 5). This helps to avoid developers to call the methods encapsulated by the facade directly.

```
1 public aspect FacadePolicyEnforcement {
2     pointcut encapsulatedMethods(): call(* (Decoration ||
          RegularScreen || StringTransformer).*(..));
3     pointcut facade(): within(OutputFacade);
4     declare warning: encapsulatedMethods() && !facade(): "
          Calling encapsulated method directly";
5 }
```

Another example, the *SingletonProtocol* aspect (HANNEMANN; KICZALES, 2002), aims to compose pointcut predicates, but do not define a separated pointcut definition to a singleton construction (line 3). Extracting this piece of predicate helps to clarify the aspect.

```
1 public abstract aspect SingletonProtocol {
2     protected pointcut protectionExclusions();
3     Object around(): call((Singleton+).new(..)) &&
4     !protectionExclusions() {...}
5 }
```

If a new pointcut definition is created, a pointcut composition of *singletonCreation* and *protectionExclusions* can be used. The next example illustrates the use of the composite predicate. Note the extracted pointcut *sigletonCreation* in line 2.

```
1 public abstract aspect SingletonProtocol {
2     pointcut singletonCreation(): call((Singleton+).new(..));
3     Object around(): singletonCreation() && !
          protectionExclusions() {...}
4 }
```

B.5 One Concern per Aspect

Guideline. Design aspects so that they provide functionality to only one concern of the application. If this aspect deals with more than one concern, try to divide it in two or more aspects or classes, maybe forming a generalization hierarchy.

Motivation. When an aspect handles more than one concern, it should be divided into smaller aspects, each one responsible for a single concern. This often happens with advices with diverging purposes or with attributes and inter-type declarations without connection with the rest of the aspect.

Example. Consider a *Debug* aspect (part of an example named *Space War* - a spaceship and asteroids game (HILSDALE; KICZALES, 2001)), which defines advices dealing with different concerns simultaneously. This aspect collects points regarding user interface modification (line 2), regarding changes in the registry contents (line 3), and regarding ship collisions (line 4), among other concerns omitted in the example. Although all of these features are related to system debugging, they can be divided in several aspects, each one with a different perspective on debugging.

```
1 aspect Debug {
2   after() returning (SWFrame frame): call(SWFrame+.new(..))
        {...}
```

```
3    after(Registry registry) returning : target(registry) && (
         call(void register(..)) || call(void unregister(..)))
         {...}
4    after(Ship ship, SpaceObject obj) returning : call(void Ship
         .handleCollision(SpaceObject)) && target(ship) && args(
         obj){...}
5  }
```

Another possibility is to make the different extracted aspects inherit from the same super-class (or super-aspect), which can be the *Debug* aspect itself. The following example uses a sub-aspect of *Debug* containing an advice responsible for manipulating the debugging of ship collisions (line 2).

```
1  aspect Collision extends Debug{
2     after(Ship ship, SpaceObject obj) returning: call(void
          Ship.handleCollision(SpaceObject)) && target(ship) &&
          args(obj){...}
3    }
```

Defining a *Collision* aspect enables the developer to separate the debugging responsibilities, focusing, in this case, only in the collision specific requirements. This separation also makes easier to reuse the *Debug* aspect, since it contains only the basic debugging functions.

B.6 Discussion

The use of these guidelines can avoid the occurrence of *anonymous pointcut definitions* and several occurrences of the *double personality*, *lazy aspects*, *code duplication* and *divergent changes* shortcomings (PIVETA et al., 2005)(PIVETA et al., 2006a).

Anonymous pointcut definitions can be avoided using the following guidelines: *use named pointcuts* and *favour pointcut composition*. The *double personality* and *lazy aspect* shortcomings can be avoided using the *one concern per aspect* guideline. *Code duplications* can be diminished by using *abstract aspects* and *named pointcuts*. The *favour pointcut composition* guideline can be used to overcome occurrences of *divergent changes*.

For the first guideline (*use abstract aspects*), it is not necessary that an abstract aspect is created for every aspect in the application, just as not every pointcut have necessarily to be defined by an abstract aspect. Abstract aspects are the core of aspect reuse. They are heavily used in the aspect-oriented design patterns and they allow the definition of a reusable implementation of several design patterns.

The use of this guideline is exemplified by design patterns implemented in AspectJ (HANNEMANN; KICZALES, 2002), by access and authentication mechanisms (VANHAUTE; De Win; De Decker, 2001), and by the implementation of distribution and persistence components (SOARES; LAUREANO; BORBA, 2002). Vanhaute et al. (VANHAUTE; De Win; De Decker, 2001) state that the use of abstract pointcuts and inheritance between aspects helps in the generalization required for implementing aspect-based frameworks. Some further examples of the usage of abstract aspects appear in the implementation of the following patterns: *Chain Of Responsibility*, *Command*, *Mediator*, *Observer*, *Flyweight* and *Memento*, among others. Usually, the user binds the roles in the patterns using aspect inheritance.

The use of *named pointcuts* can be found in several aspects in the pattern library

(HANNEMANN; KICZALES, 2002). They are used as a key resource to the definition of reusable aspects. Examples of aspects using named pointcuts are: *ChainOfResponsibilityProtocol*, *CommandProtocol*, *MediatorProtocol*, *Decorator* aspects and *ObserverProtocol*. Mahrenholz et al. (MAHRENHOLZ; SPINCZYK; SCHRODER-PREIKSCHAT, 2002) remind that the use of named pointcuts allows one to create formal arguments (parameters) for the occurrences referred by the predicate defined in the pointcut. Alwis et al. (ALWIS et al., 2000) assert that using relevant names for pointcuts can promote the reuse of the aspects. The names of the pointcuts should describe, at high level, which kinds of operations (or other more complex join points) fit the pointcuts context, instead of the working details of the operation.

The use of *semantic based pointcuts* provides mechanisms to use annotations when there is the need to group different structures in a set of classes or a set of aspects. This grouping is performed in a way to consider semantic aspects above syntactic ones. Even the application of renaming refactoring patterns can cause pointcuts to affect a different set of join points.

The *favour pointcut composition* guideline contributes to the clarity of the specification, and at the same time allows pointcuts to be reused individually. Following this guideline can help avoiding shortcomings with respect to *divergent changes* and *code duplication* in aspects (PIVETA et al., 2005).

The *one concern per aspect* guideline can be used when, in modelling for example, the concerns encapsulated by an aspect deal with an inheritance relationship. This should be made explicit by moving duplicated members to a super-aspect. After applying a set of refactoring patterns, aspects with few responsibilities can appear. Aspects without sufficient responsibilities can be merged with another aspect or class. Empty aspects or aspects that do not contain advices, pointcuts or inter-type declarations can be converted to classes or merged with other aspects.

B.7 Conclusions

This appendix provides a set of guidelines to help on avoiding the occurrence of shortcomings in aspect-oriented software. These guidelines are exemplified and discussed using a set of AspectJ examples from different sources. The guidelines can assist software designers in presence of crosscutting concerns and complement other guidelines for aspect-oriented design (CHAVEZ; LUCENA, 2004; ALWIS et al., 2000; HANENBERG; UNLAND, 2001). Several shortcomings can be minimized following some basic design guidelines.

Although the guidelines discussed in this appendix are expressed in examples of a specific language, they can be adapted to other aspect-oriented languages. Some guidelines can be used directly, whereas some of them are not available in a chosen language, for example. As the AspectJ model is the basis for several aspect languages, it can be seen as a good starting point to the definition of shortcomings and design guidelines for a larger set of aspect-oriented software.

APPENDIX C AN ANALYTICAL EVALUATION FOR A SET OF ASPECT-ORIENTED METRICS

This appendix shows an analytical evaluation of the set of aspect-oriented metrics described in Chapter 9.

Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) state that several researchers recommend properties that software metrics should possess to increase their usefulness. They choose the Weyuker's criteria (WEYUKER, 1988) to evaluate a set of size and coupling metrics for object-oriented software because it is a widely known formal analytical approach and also because her formal analytical approach subsumes most of the earlier, less well-defined and informal properties.

Since this research is evaluating metrics adapted from Chidamber and Keremer metrics (CHIDAMBER; KEMERER, 1994) (Chapter 9), the same criteria to evaluate the original metrics is used. Note that the criteria are paradigm-independent (WEYUKER, 1988) and can be used to evaluate both object-oriented and aspect-oriented software. The two additional metrics (crosscutting degree of an aspect (*cda*) and coupling on advice execution (*cae*)) are also evaluated using the same criteria. As the Weyuker's criteria are also used to evaluate coupling and cohesion object-oriented metrics, there are no issues associated with the use of the criteria to evaluate these additional coupling metrics.

The criteria are summarised and expressed using predicate logic as follows (WEYUKER, 1988). For all the properties, let us consider the modules [1] $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and the metric μ:

- *Non-coarseness (property 1):* This property verifies that the metric value can be different among modules, otherwise the metric is not meaningful. Given $\mathcal{A}$ and $\mathcal{B}$, the predicate $\forall \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ must hold.

- *Non-uniqueness (property 2):* This property expresses that two different modules can have the same value for the metric (i.e. the modules are equally complex). Given $\mathcal{A}$ and $\mathcal{B}$, the predicate $\exists\ \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) = \mu(\mathcal{B})$ must hold.

- *Design details are important (property 3):* This property leads to the notion that different design alternatives can produce different values for the metrics. Given $\mathcal{A}$ and $\mathcal{B}$ providing the same functionality, the predicate $\mu(\mathcal{A}) = \mu(\mathcal{B})$ is not necessarily true.

- *Monotonicity (property 4):* This property states that the value of the metric for the composition of two modules can never be less than the metric values of each

[1] When applicable, the generic term *module* is used to denote a class or aspect.

individual module. The predicate $\forall \mathcal{A}, \mathcal{B}\ \mu(\mathcal{A}) \leq \mu(\mathcal{A}+\mathcal{B}) \wedge \mu(\mathcal{B}) \leq \mu(\mathcal{A}+\mathcal{B})$ must hold, where $\mathcal{A}+\mathcal{B}$ denotes the composition between $\mathcal{A}$ and $\mathcal{B}$.

- *Non-equivalence of interaction (property 5):* This property considers that the composition of $\mathcal{A}$ and $\mathcal{B}$ can result in different values for the same metric that the composition of $\mathcal{A}$ and $\mathcal{C}$. In this case, $\mu(\mathcal{A}) = \mu(\mathcal{B})$ does not imply that $\mu(\mathcal{A}+\mathcal{C}) = \mu(\mathcal{B}+\mathcal{C})$.
- *Interaction increases complexity (property 6):* This property states that when two modules are combined, the metric value can increase. $\exists\, \mathcal{A}, \mathcal{B}$ such as: $\mu(\mathcal{A}+\mathcal{B}) > \mu(\mathcal{A}) + \mu(\mathcal{B})$.

These properties (one to six) are used for the analytical evaluation of each metric in the next section.

C.1 Lines of Code

Consider an aspect $\mathcal{A}$ and an aspect $\mathcal{B}$ that is an exact copy of $\mathcal{A}$ plus an additional field. The value of $locc(\mathcal{B}) = locc(\mathcal{A}) + 1$ and Property 1 is satisfied, as the predicate $\forall \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ holds. Property 2 is satisfied, as for each $\mathcal{A}$, an exact copy $\mathcal{B}$ can be created, and therefore the predicate $\exists\ \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) = \mu(\mathcal{B})$ holds. The $locc$ of each construction in a class is a design decision and it is not determined by the functionality of the aspect, therefore Property 3 is satisfied.

The $locc$ of the composition of $\mathcal{A}$ and $\mathcal{B}$ can be defined as $locc(\mathcal{A}+\mathcal{B}) = locc(\mathcal{A}) + locc(\mathcal{B}) - \kappa$, where κ is the number of duplicated lines of code in any constructions of $\mathcal{A}$ or $\mathcal{B}$, such as methods, advices or fields that are common to $\mathcal{A}$ and $\mathcal{B}$. The value of κ can vary from $[0, min(locc(\mathcal{A}), locc(\mathcal{B}))]$. Either if $(\kappa = 0)$ or $(\kappa = min(locc(\mathcal{A}), locc(\mathcal{B})))$, $locc(\mathcal{A}+\mathcal{B}) \geq locc(\mathcal{A})$ and $locc(\mathcal{A}+\mathcal{B}) \geq locc(\mathcal{B})$. Therefore, Property 4 is satisfied.

Let $locc(\mathcal{A}) = locc(\mathcal{B})$. Consider an aspect $\mathcal{C}$, containing κ lines of code in common with $\mathcal{A}$ and $\kappa+1$ lines of code in common with $\mathcal{B}$. Therefore $locc(\mathcal{A}+\mathcal{C}) \neq locc(\mathcal{B}+\mathcal{C})$ as $locc(\mathcal{A}+\mathcal{C}) = locc(\mathcal{A}) + locc(\mathcal{C}) - \kappa$ and $locc(\mathcal{B}+\mathcal{C}) = locc(\mathcal{B}) + locc(\mathcal{C}) - \kappa + 1$. In this case, Property 5 is satisfied. Consider κ as the number of common lines of code between $\mathcal{A}$ and $\mathcal{B}$. As $locc(\mathcal{A}) + locc(\mathcal{B}) - \kappa \leq locc(\mathcal{A}) + locc(\mathcal{A})$ for all $\mathcal{A}$ and $\mathcal{B}$, Property 6 is not satisfied. Failing to satisfy this property implies that the metric values can increase if an aspect or class is divided in more aspects or classes. Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) claim that this property may not be an essential feature for object-oriented software design complexity metrics and not satisfying it can be seen as beneficial in object-oriented software. Their interpretation can be corroborated for aspect-oriented software.

C.2 Number of Operations in Module

Consider two identical aspects $\mathcal{A}$ and $\mathcal{B}$. If an advice is added to $\mathcal{B}$ then $nom(\mathcal{A}) = nom(\mathcal{B}) - 1$. Property 1 is satisfied (the predicate $\forall \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ holds). Property 2 is satisfied, as for all aspect $\mathcal{A}$ one can create a class with the same number of operations (by cloning the class, for example). The number of advices, inter-type declarations, declare constructions and methods is a design decision and is not dependent of the functionality of the aspect, therefore Property 3 is satisfied.

The nom of the composition of $\mathcal{A}$ and $\mathcal{B}$ can be defined as $nom(\mathcal{A}+\mathcal{B}) = nom(\mathcal{A}) + nom(\mathcal{B}) - \omega$, where ω is the number of common operations (methods,

advices, inter-type declarations) between $\mathcal{A}$ and $\mathcal{B}$. The value of ω can vary from $[0, min(nom(\mathcal{A}), nom(\mathcal{B}))]$. Either if $(\omega = 0)$ or $(\omega = min(nom(\mathcal{A}), nom(\mathcal{B})))$, $nom(\mathcal{A} + \mathcal{B}) \geq nom(\mathcal{A})$ and $nom(\mathcal{A} + \mathcal{B}) \geq nom(\mathcal{B})$. Therefore, Property 4 is satisfied.

Let $nom(\mathcal{A}) = nom(\mathcal{B})$. Consider a $\mathcal{C}$ aspect, containing ω operations in common with $\mathcal{A}$ and $\omega+1$ operations in common with $\mathcal{B}$. Therefore $nom(\mathcal{A}+\mathcal{C}) \neq nom(\mathcal{B}+\mathcal{C})$ as $nom(\mathcal{A}+\mathcal{C}) = nom(\mathcal{A}) + nom(\mathcal{C}) - \omega$ and $nom(\mathcal{B}+\mathcal{C}) = nom(\mathcal{B}) + nom(\mathcal{C}) - \omega + 1$. In this case, Property 5 is satisfied. Consider ω as the number of common operations between $\mathcal{A}$ and $\mathcal{B}$. For all $\mathcal{A}$ and $\mathcal{B}$, $nom(\mathcal{A}) + nom(\mathcal{B}) - \omega \leq nom(\mathcal{A}) + nom(\mathcal{B})$ and therefore Property 6 is not satisfied (the effects of not satisfying this property are the same of those for the $locc$ metric). Section C.7 discusses more details about not satisfying this property.

C.3 Depth of Inheritance Tree

Consider two aspects $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{B}$ is a sub-aspect of $\mathcal{A}$. In this case, the value of $dit(\mathcal{B}) = dit(\mathcal{A}) + 1$. Property 1 is satisfied, as the predicate $\forall \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ holds. The dit for any sibling of $\mathcal{B}$ is also $dit(\mathcal{A}) + 1$, so Property 2 is satisfied. Property 3 is satisfied as the use of inheritance mechanisms is a design dependent issue and is independent of the aspects functionality.

The combination of $\mathcal{A}$ and $\mathcal{B}$, in terms of inheritance, depends on the following situations: (a) $\mathcal{A}$ and $\mathcal{B}$ are super and sub-aspects (b) $\mathcal{A}$ and $\mathcal{B}$ are siblings and (c) $\mathcal{A}$ and $\mathcal{B}$ are unrelated in the inheritance tree. For situation (a), if $\mathcal{A}$ and $\mathcal{B}$ are composed, the $dit(\mathcal{A} + \mathcal{B})$ is equal to the dit of the super-aspect. In this case, the predicate $dit(\mathcal{A}+\mathcal{B}) \geq dit(superAspect) \wedge dit(\mathcal{A}+\mathcal{B}) \geq dit(subAspect)$ does not hold, and therefore, for this specific case, Property 4 is not satisfied. For situation (b), $dit(\mathcal{A}) = dit(\mathcal{B})$ and therefore $dit(\mathcal{A}+\mathcal{B}) \geq dit(\mathcal{A}) \wedge dit(\mathcal{A}+\mathcal{B}) \geq dit(\mathcal{A})$ holds. In this case, Property 4 is satisfied. For situation (c), if the direct common ancestor of $\mathcal{A}$ and $\mathcal{B}$ is the super-aspect or super-class of $\mathcal{A}$, the combination of both aspects is located in $\mathcal{B}$ actual's location. In this case, Property 4 is satisfied, as $dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A}) \wedge dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A})$. If the common ancestor of both aspects is not a direct ancestor of both $\mathcal{A}$ and $\mathcal{B}$, there is the need to use multiple inheritance to address the situation (which is not a common feature in aspect-oriented languages). For a discussion of the dit metric for object-oriented software, refer to Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994).

The dit for aspects fails to satisfy Property 4 only when two aspects are in a parent-descendent relationship. The same situation happens with the metrics for object-oriented software (CHIDAMBER; KEMERER, 1994). Not satisfying this property does not invalidate the use of dit as a metric to assess the use of inheritance mechanisms (the same occurs to the metric for object-oriented software).

Let $dit(\mathcal{A}) = dit(\mathcal{B})$. Let $\mathcal{C}$ be a sub-aspect of $\mathcal{A}$. As $dit(\mathcal{A} + \mathcal{C}) = dit(\mathcal{A})$ and $dit(\mathcal{B} + \mathcal{C}) = dit(\mathcal{A}) + 1$, the predicate $dit(\mathcal{A} + \mathcal{C}) \neq dit(\mathcal{B} + \mathcal{C})$ holds and therefore Property 5 is satisfied. Considering that the $dit(\mathcal{A} + \mathcal{B})$ is equal to $max(dit(\mathcal{A}), dit(\mathcal{B}))$, $dit(\mathcal{A} + \mathcal{B}) \leq dit(\mathcal{A} + \mathcal{B})$ for all $\mathcal{A}$ and $\mathcal{B}$. Property 6 is not satisfied (the effects of not satisfying this property are the same of those for the $locc$ metric). Section C.7 discusses more details about this.

C.4 Number of Children

Let $\mathcal{A}$ and $\mathcal{B}$ be leaves and $\mathcal{C}$ be the root of an an inheritance tree. Property 1 is satisfied as $noc(\mathcal{C}) \neq noc(\mathcal{A})$. As $\mathcal{A}$ and $\mathcal{B}$ are leaves, they both have $noc = 0$, so Property 2 is satisfied. Also, Property 3 is satisfied as the noc of an aspect is a design issue and is independent of the functionality. Let $\mathcal{B}$ be the only sub-aspect or sub-class of $\mathcal{A}$. If $\mathcal{A}$ and $\mathcal{B}$ are combined the value of $noc(\mathcal{A}+\mathcal{B}) = 0$ and the predicate $noc(\mathcal{A}+\mathcal{B}) \geq noc(\mathcal{A}) \wedge noc(\mathcal{A}+\mathcal{B}) \geq noc(\mathcal{A})$ does not hold as $noc(\mathcal{A}+\mathcal{B}) < noc(\mathcal{A})$. Therefore, Property 4 is not satisfied. As happens with the dit metric, the noc for aspects fails to satisfy Property 4 when two aspects are in a parent-descendent relationship. As discussed before, not following this property does not invalidate the use of noc as a metric to assess the use of inheritance mechanisms. This is discussed with more details in Section C.7.

Now, let $noc(\mathcal{A}) = noc(\mathcal{B})$ and let $\mathcal{C}$ be a sub-aspect of $\mathcal{A}$. As $noc(\mathcal{A}+\mathcal{C}) = noc(\mathcal{A}) + noc(\mathcal{C}) - 1$, $noc(\mathcal{B}+\mathcal{C}) = noc(\mathcal{B}) + noc(\mathcal{C})$ and $noc(\mathcal{A}) = noc(\mathcal{B})$, the predicate $noc(\mathcal{A}+\mathcal{C}) \neq noc(\mathcal{B}+\mathcal{C})$ holds and Property 5 is satisfied. Considering that the maximum value of $noc(\mathcal{A}+\mathcal{B})$ is equal to $noc(\mathcal{A}) + noc(\mathcal{B})$ for all $\mathcal{A}$ and $\mathcal{B}$, the Property 6 is not satisfied (the effects of not satisfying this property are the same of those for the $locc$ metric). Section C.7 discusses additional details.

C.5 Crosscutting Degree of an Aspect

Consider two aspects $\mathcal{A}$ and $\mathcal{B}$. It is always possible to create a new module $\mathcal{C}$ and insert an inter-type declaration in $\mathcal{B}$ module, for example. Property 1 is satisfied, as the predicate $\forall\mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ holds. Two empty aspects $\mathcal{A}$ and $\mathcal{B}$ have equal values to the cda metric. In this case, Property 2 is satisfied. Property 3 is also satisfied, as the number of modules affected by aspects is dependent of design decisions and not of functionality.

If the aspects $\mathcal{A}$ and $\mathcal{B}$ are combined, the $cda(\mathcal{A}+\mathcal{B}) = cda(\mathcal{A}) + cda(\mathcal{B}) - \nu$, where ν is the number of common modules affected by $\mathcal{A}$ and $\mathcal{B}$. In this case, ν is given by a value in the interval $[0, min(cda(\mathcal{A}), cda(\mathcal{B}))]$, if $\nu = 0$, $cda(\mathcal{A}+\mathcal{B}) = cda(\mathcal{A})+cda(\mathcal{B})$. Property 4 is satisfied. If $\nu = min(cda(\mathcal{A}), cda(\mathcal{B}))$ then $cda(\mathcal{A}+\mathcal{B}) = max(cda(\mathcal{A}), cda(\mathcal{B}))$ and Property 4 remains satisfied.

Consider that $cda(\mathcal{A}) = cda(\mathcal{B})$ and let $\mathcal{C}$ be another aspect. Consider that the number of affected modules in $\mathcal{C}$ that are common with $\mathcal{A}$ is υ and in common with $\mathcal{B}$ is ω and also that $\upsilon \neq \omega$. As $cda(\mathcal{A}+\mathcal{C}) = cda(\mathcal{A})+cda(\mathcal{C})-\upsilon$ and $cda(\mathcal{B}+\mathcal{C}) = cda(\mathcal{B})+cda(\mathcal{C})-\omega$, the predicate $cda(\mathcal{A}+\mathcal{C}) \neq cda(\mathcal{B}+\mathcal{C})$ holds and therefore Property 5 is satisfied. Consider that the number of common modules affected by advices, inter-type method declarations or inter-type constructor declarations between two aspects $\mathcal{A}$ and $\mathcal{B}$ is given by ν. For all $\mathcal{A}$ and $\mathcal{B}$, $cda(\mathcal{A}) + cda(\mathcal{B}) - \nu \leq cda(\mathcal{A}) + cda(\mathcal{B})$ and therefore Property 6 is not satisfied (the implications are the same of those for the $locc$ metric).

C.6 Coupling on Advice Execution

Consider two equal aspects $\mathcal{A}$ and $\mathcal{B}$. It is always possible to create a new aspect $\mathcal{C}$ that affects only $\mathcal{B}$, for example. In this case, $cae(\mathcal{A}) = cae(\mathcal{B}) - 1$. Property 1 is satisfied, as $\forall\mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$. Property 2 is also satisfied, as two unaffected classes have the same value for the cae metric. The number of aspects that affects a class is design dependent and not mandated by the functionality of the classes and aspects, therefore

Property 3 is also satisfied.

The value of $cae(\mathcal{A} + \mathcal{B}) = cae(\mathcal{A}) + cae(\mathcal{B}) - \theta$, where θ is the number of common modules that affect $\mathcal{A}$ and $\mathcal{B}$. In this case, θ is within the interval [0, min(cae($\mathcal{A}$), cae($\mathcal{B}$))]. If $\theta = 0$ then $cda(\mathcal{A} + \mathcal{B}) = cae(\mathcal{A}) + cae(\mathcal{B})$, and therefore Property 4 is satisfied. If $\theta = min(cae(\mathcal{A}), cae(\mathcal{B}))$ then $cda(\mathcal{A} + \mathcal{B}) = max(cae(\mathcal{A}), cae(\mathcal{B}))$ and Property 4 remains satisfied.

Consider that $cae(\mathcal{A}) = cae(\mathcal{B})$ and let $\mathcal{C}$ be another module. Consider that the number of aspects affecting $\mathcal{C}$ that are common with the aspects that affect $\mathcal{A}$ is υ and in common with $\mathcal{B}$ is ω and also that $\upsilon \neq \omega$ As $cae(\mathcal{A} + \mathcal{C}) = cae(\mathcal{A}) + cae(\mathcal{C}) - \upsilon$ and $cae(\mathcal{B} + \mathcal{C}) = cae(\mathcal{B}) + cae(\mathcal{C}) - \omega$, the predicate $cae(\mathcal{A} + \mathcal{C}) \neq cae(\mathcal{B} + \mathcal{C})$ holds and therefore Property 5 is satisfied. Let θ be the number of common modules affecting two modules $\mathcal{A}$ and $\mathcal{B}$. For all $\mathcal{A}$ and $\mathcal{B}$, $cae(\mathcal{A}) + cae(\mathcal{B}) - \theta \leq cae(\mathcal{A}) + cae(\mathcal{B})$ and therefore Property 6 is not satisfied (the effects of not satisfying this property are the same of those for the $locc$ metric).

C.7 Conclusions

Regarding the analytical results, the metrics described here satisfy most of the properties analysed, except for Property 6. However, this failure implies that the metric values can increase if an aspect or class is divided in more aspects or classes. Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994) claim that this property may not be an essential feature for object-oriented software design complexity metrics and not satisfying can be seen as beneficial in object-oriented software. The same occurs for aspect-oriented software. As happens with the metrics for object-oriented software (CHIDAMBER; KEMERER, 1994), the dit and noc for aspects fail to satisfy Property 4 only when two aspects are in a parent-descendent relationship. Not following this property does not invalidate the use of dit and noc as metrics to assess the use of inheritance mechanisms (the same occurs to the object-oriented version of these metrics).

The analytical evaluation of the selected metrics against established criteria for validity showed that the aspect-oriented adapted metrics satisfy the criteria originally satisfied by the object-oriented metrics, which means that they can be used to assess aspect-oriented software and provide comparable results.

APPENDIX D COMPUTING IMPACT FUNCTIONS FOR PULL UP ADVICE IN THE GLASSBOX INSPECTOR

This appendix details the computation of 15 impact functions, simulating the application of the *Pull Up Advice* refactoring pattern in the Glassbox Inspector.

D.1 Super-Aspects

Three super-aspects are used in the example and are named as follows:

$$\begin{aligned} \alpha_1 &= AbstractResourceMonitor \\ \alpha_2 &= AbstractXMLProcessingMonitor \\ \alpha_3 &= AbstractRequestMonitor \end{aligned}$$

The metric values needed to compute the impact functions (as described in Chapter 7) is shown as follows.

D.1.1 AbstractResourceMonitor

$$\begin{aligned} \alpha_1 &= AbstractResourceMonitor \\ cda(\alpha_1) &= 0,\ \ cae(\alpha_1) = 3 \\ \mathcal{A}_{\alpha_1} &= \{\},\ \ \mathcal{MA}_{\alpha_1} = \{\} \\ \mathcal{E}_{\alpha_1} &= \{JmxManagement, SimpleConfig, \\ &\quad ErrorHandling\} \\ \mathcal{ME}_{\alpha_1} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\ &\quad (ErrorHandling, 1)\} \end{aligned}$$

D.1.2 AbstractXMLProcessingMonitor

$$
\begin{aligned}
\alpha_2 &= AbstractXMLProcessingMonitor \\
cda(\alpha_2) &= 0, cae(\alpha_2) = 3 \\
\mathcal{A}_{\alpha_2} &= \{\}, \ \mathcal{MA}_{\alpha_2} = \{\} \\
\mathcal{E}_{\alpha_2} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling\} \\
\mathcal{ME}_{\alpha_2} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 3)\}
\end{aligned}
$$

D.1.3 AbstractRequestMonitor

$$
\begin{aligned}
\alpha_3 &= AbstractRequestMonitor \\
cda(\alpha_3) &= 1, cae(\alpha_3) = 4 \\
\mathcal{A}_{\alpha_3} &= \{RequestContext\} \\
\mathcal{MA}_{\alpha_3} &= \{(RequestContext, 1)\} \\
\mathcal{E}_{\alpha_3} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling\} \\
\mathcal{ME}_{\alpha_3} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 22)\}
\end{aligned}
$$

D.2 Sub-Aspects

Six aspects are the sub-aspects (the β aspects):

$$
\begin{aligned}
\beta_1 &= JDBCConnectionMonitor \\
\beta_2 &= JDBCStatementMonitor \\
\beta_3 &= RemoteCallMonitor \\
\beta_4 &= JaxmCallMonitor \\
\beta_5 &= AbstractXMLCallMonitor \\
\beta_6 &= AbstractOperationMonitor
\end{aligned}
$$

The metric values needed to compute the impact functions are shown in the following sub-sections.

D.2.1 JDBCConnectionMonitor

$$
\begin{aligned}
\beta_1 &= JDBCConnectionMonitor \\
locc(\beta_1) &= 76,\ \ nom(\beta_1) = 10,\ \ cda(\beta_1) = 0,\ \ cae(\beta_1) = 5 \\
\mathcal{A}_{\beta_1} &= \{\} \\
\mathcal{MA}_{\beta_1} &= \{\} \\
\mathcal{E}_{\beta_1} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling, TrackParents, \\
&\quad ServletMonitor\} \\
\mathcal{ME}_{\beta_1} &= \{(JmxManagement\,1), (SimpleConfig\,1), \\
&\quad (ErrorHandling\,15), (TrackParents\,2), \\
&\quad (ServletMonitor\,1)\}
\end{aligned}
$$

D.2.2 JDBCStatementMonitor

$$
\begin{aligned}
\beta_2 &= JDBCStatementMonitor \\
locc(\beta_2) &= 92,\ \ nom(\beta_2) = 12,\ \ cda(\beta_2) = 0,\ \ cae(\beta_2) = 5 \\
\mathcal{A}_{\beta_2} &= \{\} \\
\mathcal{MA}_{\beta_2} &= \{\} \\
\mathcal{E}_{\beta_2} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling, TrackParents, \\
&\quad ServletMonitor\} \\
\mathcal{ME}_{\beta_2} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 11), (TrackParents, 2), \\
&\quad (ServletMonitor, 1)\}
\end{aligned}
$$

D.2.3 RemoteCallMonitor

$$
\begin{aligned}
\beta_3 &= RemoteCallMonitor \\
locc(\beta_3) &= 34,\ \ nom(\beta_3) = 6,\ \ cda(\beta_3) = 0,\ \ cae(\beta_3) = 4 \\
\mathcal{A}_{\beta_3} &= \{\} \\
\mathcal{MA}_{\beta_3} &= \{\} \\
\mathcal{E}_{\beta_3} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\beta_3} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 2), (TrackParents, 2)\}
\end{aligned}
$$

D.2.4 JaxmCallMonitor

$$
\begin{aligned}
\beta_4 &= JaxmCallMonitor \\
locc(\beta_4) &= 18,\ \ nom(\beta_4) = 3,\ \ cda(\beta_4) = 0,\ \ cae(\beta_4) = 4 \\
\mathcal{A}_{\beta_4} &= \{\} \\
\mathcal{MA}_{\beta_4} &= \{\} \\
\mathcal{E}_{\beta_4} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\beta_4} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 1), (TrackParents, 1)\}
\end{aligned}
$$

D.2.5 AbstractXMLCallMonitor

$$
\begin{aligned}
\beta_5 &= AbstractXMLCallMonitor \\
locc(\beta_5) &= 23,\ \ nom(\beta_5) = 3,\ \ cda(\beta_5) = 0,\ \ cae(\beta_5) = 4 \\
\mathcal{A}_{\beta_5} &= \{\} \\
\mathcal{MA}_{\beta_5} &= \{\} \\
\mathcal{E}_{\beta_5} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\beta_5} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 2), (TrackParents, 1)\}
\end{aligned}
$$

D.2.6 AbstractOperationMonitor

$$
\begin{aligned}
\beta_6 &= RemoteCallMonitor \\
locc(\beta_6) &= 100,\ \ nom(\beta_6) = 20,\ \ cda(\beta_6) = 1,\ \ cae(\beta_6) = 4 \\
\mathcal{A}_{\beta_6} &= \{AbstractOperationMonitor\} \\
\mathcal{MA}_{\beta_6} &= \{(AbstractOperationMonitor, 4)\} \\
\mathcal{E}_{\beta_6} &= \{JmxManagement, SimpleConfig, \\
&\quad ErrorHandling, AbstractOperationMonitor\} \\
\mathcal{ME}_{\beta_6} &= \{(JmxManagement, 1), (SimpleConfig, 1), \\
&\quad (ErrorHandling, 12), (AbstractOperationMonitor, 4)\}
\end{aligned}
$$

D.3 Advices

In the β aspects, there are 15 different advices which can be pulled up to the super-aspects:

$$
\begin{aligned}
\rho_1 &= around(DataSource) \\
\rho_2 &= around(String) \\
\rho_3 &= before(Statement, String) \\
\rho_4 &= around(Statement) \\
\rho_5 &= afterreturning(Connection) \\
\rho_6 &= around(String) \\
\rho_7 &= around(Object) : remote \ldots \\
\rho_8 &= around(Object) : jaxRPC \ldots \\
\rho_9 &= around(Object, Object, Object) \\
\rho_{10} &= around(Node) \\
\rho_{11} &= afterreturning(Object) \\
\rho_{12} &= around(Object) : class \ldots \\
\rho_{13} &= around(Object) : methodSig \ldots \\
\rho_{14} &= around(Object) : methodNameCon \ldots \\
\rho_{15} &= around(Object) : methodCon \ldots
\end{aligned}
$$

D.3.1 ρ_1 - around(DataSource)

$$
\begin{aligned}
\rho_1 &= around(DataSource) \\
locc(\rho_1) &= 10 \\
wom(\rho_1) &= 1 \\
cda(\rho_1) &= 0 \\
cae(\rho_1) &= 2 \\
\mathcal{A}_{\rho_1} &= \{\} \\
\mathcal{MA}_{\rho_1} &= \{\} \\
\mathcal{E}_{\rho_1} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_1} &= \{(ErrorHandling, 3), (TrackParents, 1)\}
\end{aligned}
$$

```
1   Connection around(final DataSource dataSource) :
2    dataSourceConnectionCall(dataSource) && !
        nestedConnectionCall() &&
3    monitorEnabled()  {
4    RequestContext requestContext = new
        ConnectionRequestContext() {
5       public Object doExecute() {
6          accessingConnection(dataSource);
```

```
7            Connection connection = proceed(dataSource);
8            return addConnection(connection);
9          }
10       };
11       return (Connection) requestContext.execute();
12     }
```

D.3.2 ρ_2 - around(String)

$$
\begin{aligned}
\rho_2 &= around(DataSource) \\
locc(\rho_2) &= 10 \\
wom(\rho_2) &= 1 \\
cda(\rho_2) &= 0 \\
cae(\rho_2) &= 2 \\
\mathcal{A}_{\rho_2} &= \mathcal{MA}_{\rho_2} = \{\} \\
\mathcal{E}_{\rho_2} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_2} &= \{(ErrorHandling, 3), (TrackParents, 1)\}
\end{aligned}
$$

```
1    Connection around(final String url) :
2       directConnectionCall(url) && !nestedConnectionCall()
3       && monitorEnabled() {
4       RequestContext requestContext = new
           ConnectionRequestContext() {
5         public Object doExecute() {
6           accessingConnection(url);
7           Connection connection = proceed(url);
8           return addConnection(connection);
9         }
10      };
11      return (Connection)requestContext.execute();
12   }
```

D.3.3 ρ_3 - before(Statement,String)

$$
\begin{aligned}
\rho_3 &= before(Statement, String) \\
locc(\rho_3) &= 4 \\
wom(\rho_3) &= 0 \\
cda(\rho_3) &= 0 \\
cae(\rho_3) &= 1 \\
\mathcal{A}_{\rho_3} &= \mathcal{MA}_{\rho_3} = \{\} \\
\mathcal{E}_{\rho_3} &= \{ErrorHandling\} \\
\mathcal{ME}_{\rho_3} &= \{(ErrorHandling, 1)\}
\end{aligned}
$$

```
1   before(Statement statement, String sql):
2       statementExec(statement) && args(sql, ..) {
3       sql = stripAfterWhere(sql);
4       setUpStatement(statement, sql);
5   }
```

D.3.4 ρ_4 - around(Statement)

$$
\begin{aligned}
\rho_4 &= around(Statement) \\
locc(\rho_4) &= 10 \\
wom(\rho_4) &= 1 \\
cda(\rho_4) &= 0 \\
cae(\rho_4) &= 2 \\
\mathcal{A}_{\rho_4} &= \mathcal{MA}_{\rho_4} = \{\} \\
\mathcal{E}_{\rho_4} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_4} &= \{(ErrorHandling, 2), (TrackParents, 1)\}
\end{aligned}
$$

```
1   Object around(final Statement statement) :
2     statementExec(statement) && monitorEnabled()  {
3      RequestContext requestContext = new
           StatementRequestContext() {
4        public Object doExecute() {
5          curStatement = statement;
6          return proceed(statement);
7        }
8        protected String getRequestType() { return "execute"; }
9      };
10     return requestContext.execute();
11  }
```

D.3.5 ρ_5 - after returning(Connection)

$$
\begin{aligned}
\rho_5 &= afterreturning(Connection) \\
locc(\rho_5) &= 6 \\
wom(\rho_5) &= 0 \\
cda(\rho_5) &= 0 \\
cae(\rho_5) &= 1 \\
\mathcal{A}_{\rho_5} &= \mathcal{MA}_{\rho_5} = \{\} \\
\mathcal{E}_{\rho_5} &= \{ErrorHandling\} \\
\mathcal{ME}_{\rho_5} &= \{(ErrorHandling, 1)\}
\end{aligned}
$$

```
1    after(Connection connection) returning (Statement statement)
         :
2        callCreateStatement(connection) {
3          synchronized (JdbcStatementMonitor.this) {
4            statementCreators.put(statement, connection);
5          }
6    }
```

D.3.6 ρ_6 - around(String)

$$
\begin{aligned}
\rho_6 &= around(String) \\
locc(\rho_6) &= 11 \\
wom(\rho_6) &= 2 \\
cda(\rho_6) &= 0 \\
cae(\rho_6) &= 2 \\
\mathcal{A}_{\rho_6} &= \mathcal{MA}_{\rho_6} = \{\} \\
\mathcal{E}_{\rho_6} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_6} &= \{(ErrorHandling, 3), (TrackParents, 1)\}
\end{aligned}
$$

```
1    Object around(final String sql) :
2      callCreatePreparedStatement(sql) && monitorEnabled()  {
3       RequestContext requestContext = new
            StatementRequestContext() {
4         public Object doExecute() {
5           curStatement = (PreparedStatement)proceed(sql);
6           setUpStatement(curStatement, sql);
7           return curStatement;
8         }
9         protected String getRequestType() { return "prepare"; }
10      };
11     return requestContext.execute();
12   }
```

D.3.7 ρ_7 - around(Object): remote...

$$
\begin{aligned}
\rho_7 &= around(Object) : remote \ldots \\
locc(\rho_7) &= 13 \\
wom(\rho_7) &= 2 \\
cda(\rho_7) &= 0 \\
cae(\rho_7) &= 2 \\
\mathcal{A}_{\rho_7} &= \mathcal{MA}_{\rho_7} = \{\} \\
\mathcal{E}_{\rho_7} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_7} &= \{(ErrorHandling, 2), (TrackParents, 1)\}
\end{aligned}
$$

```
1   Object around(final Object recipient) :
2     remoteProxyCall(recipient) && monitorEnabled() {
3     RequestContext requestContext = new
        ResourceRequestContext() {
4       public Object doExecute() {
5         return proceed(recipient);
6       }
7       public PerfStats lookupStats() {
8         String key = "jaxrpc:"+recipient.getClass().getName()
            +
9           "."+thisJoinPointStaticPart.getSignature().getName()
              ;
10        key = key.intern();
11        return lookupResourceStats(key);
12      }
13    };
14    return requestContext.execute();
15  }
```

D.3.8 ρ_8 - around(Object): jaxRPC...

$$
\begin{aligned}
\rho_8 &= around(Object) : jaxRPC \ldots \\
locc(\rho_8) &= 13 \\
wom(\rho_8) &= 2 \\
cda(\rho_8) &= 0 \\
cae(\rho_8) &= 2 \\
\mathcal{A}_{\rho_8} &= \mathcal{MA}_{\rho_8} = \{\} \\
\mathcal{E}_{\rho_8} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_8} &= \{(ErrorHandling, 2), (TrackParents, 1)\}
\end{aligned}
$$

```
1   Object around(final Object wsCallObj) :
2      jaxRpcClientCall(wsCallObj) && monitorEnabled() {
3       RequestContext requestContext = new
           ResourceRequestContext() {
4         public Object doExecute() {
5           return proceed(wsCallObj);
6         }
7         public PerfStats lookupStats() {
8           Call wsCall = ((Call)wsCallObj);
9           String key = wsCall.getTargetEndpointAddress()+
10           ":"+wsCall.getOperationName().toString();
11          return lookupResourceStats(key);
12        }
13      };
14      return requestContext.execute();
15    }
```

D.3.9 ρ_9 - around(Object, Object, Object)

$$
\begin{aligned}
\rho_9 &= around(Object, Object, Object) \\
locc(\rho_9) &= 11 \\
wom(\rho_9) &= 2 \\
cda(\rho_9) &= 0 \\
cae(\rho_9) &= 2 \\
\mathcal{A}_{\rho_9} &= \mathcal{MA}_{\rho_9} = \{\} \\
\mathcal{E}_{\rho_9} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_9} &= \{(ErrorHandling, 1), (TrackParents, 1)\}
\end{aligned}
$$

```
1   Object around(final Object soapConnection, final Object msg,
       final Object endPoint) :
2      jaxmCall(soapConnection, msg, endPoint) && monitorEnabled
          () {
3       RequestContext requestContext = new
           ResourceRequestContext() {
4         public Object doExecute() {
5           return proceed(soapConnection, msg, endPoint);
6         }
7         public PerfStats lookupStats() {
8           return lookupResourceStats(endPoint.toString());
9         }
10      };
11     return requestContext.execute();
12  }
```

D.3.10 ρ_{10} - around(Node)

$$
\begin{aligned}
\rho_{10} &= around(Node) \\
locc(\rho_{10}) &= 17 \\
wom(\rho_{10}) &= 2 \\
cda(\rho_{10}) &= 0 \\
cae(\rho_{10}) &= 2 \\
\mathcal{A}_{\rho_{10}} &= \{\} \\
\mathcal{MA}_{\rho_{10}} &= \{\} \\
\mathcal{E}_{\rho_{10}} &= \{ErrorHandling, TrackParents\} \\
\mathcal{ME}_{\rho_{10}} &= \{(ErrorHandling, 2), (TrackParents, 1)\}
\end{aligned}
$$

```
1  Object around(final Node node) :
2      domCall(node) && !inXmlRequest() && monitorEnabled() {
3    RequestContext requestContext = new XmlRequestContext() {
4      public Object doExecute() {
5        return proceed(node);
6      }
7      public PerfStats lookupStats() {
8        Document doc;
9        if (node instanceof Document) {
10         doc = (Document)node;
11       } else {
12         doc = node.getOwnerDocument();
13       }
14       return lookupDocumentStats(doc);
15     }
16   };
17   return requestContext.execute();
18 }
```

D.3.11 ρ_{11} - after returning(Object)

$$
\begin{aligned}
\rho_{11} &= afterreturning(Object) \\
locc(\rho_{11}) &= 3 \\
wom(\rho_{11}) &= 0 \\
cda(\rho_{11}) &= 1 \\
cae(\rho_{11}) &= 1 \\
\mathcal{A}_{\rho_{11}} &= \{AbstractOperationMonitor\} \\
\mathcal{MA}_{\rho_{11}} &= \{(AbstractOperationMonitor, 4)\} \\
\mathcal{E}_{\rho_{11}} &= \{ErrorHandling\} \\
\mathcal{ME}_{\rho_{11}} &= \{(ErrorHandling, 1)\}
\end{aligned}
$$

```
1    after(Object controller) returning (OperationRequestContext
        context):
2       cflow(adviceexecution() && args(controller, ..) && this(
           AbstractOperationMonitor)) &&
3       call(OperationRequestContext+.new(..)) {
4        context.controller = controller;
5      }
```

D.3.12 ρ_{12} - around(Object): class...

$$
\begin{aligned}
\rho_{12} &= around(Object) : class\ldots \\
locc(\rho_{12}) &= 11 \\
wom(\rho_{12}) &= 2 \\
cda(\rho_{12}) &= 0 \\
cae(\rho_{12}) &= 2 \\
\mathcal{A}_{\rho_{12}} &= \{\} \\
\mathcal{MA}_{\rho_{12}} &= \{\} \\
\mathcal{E}_{\rho_{12}} &= \{ErrorHandling, AbstractOperationMonitor\} \\
\mathcal{ME}_{\rho_{12}} &= \{(ErrorHandling, 1), (AbstractOperationMonitor, 1)\}
\end{aligned}
$$

```
1    Object around(final Object controller) :
2       classControllerExec(controller) && monitorEnabled() {
3      RequestContext rc = new OperationRequestContext() {
4        public Object doExecute() {
5          return proceed(controller);
6        }
7        protected Object getKey() {
8          return controller.getClass();
9        }
10     };
11     return rc.execute();
12   }
```

D.3.13 ρ_{13} - around(Object): methodSig...

$$
\begin{aligned}
\rho_{13} &= around(Object) : methodSig\ldots \\
locc(\rho_{13}) &= 11 \\
wom(\rho_{13}) &= 2 \\
cda(\rho_{13}) &= 0 \\
cae(\rho_{13}) &= 2 \\
\mathcal{A}_{\rho_{13}} &= \{\} \\
\mathcal{MA}_{\rho_{13}} &= \{\} \\
\mathcal{E}_{\rho_{13}} &= \{ErrorHandling, AbstractOperationMonitor\} \\
\mathcal{ME}_{\rho_{13}} &= \{(ErrorHandling, 1), (AbstractOperationMonitor, 1)\}
\end{aligned}
$$

```
1   Object around(final Object controller) :
2     methodSignatureControllerExec(controller) &&
        monitorEnabled() {
3     RequestContext rc = new OperationRequestContext() {
4       public Object doExecute() {
5         return proceed(controller);
6       }
7       protected Object getKey() {
8         return concatenatedKey(controller.getClass(),
9           thisJoinPointStaticPart.getSignature().getName());
10      }
11    };
12    return rc.execute();
13  }
```

D.3.14 ρ_{14} - around(Object): methodNameCon...

$$
\begin{aligned}
\rho_{14} &= around(Object) : methodNameCon\ldots \\
locc(\rho_{14}) &= 11 \\
wom(\rho_{14}) &= 2 \\
cda(\rho_{14}) &= 0 \\
cae(\rho_{14}) &= 2 \\
\mathcal{A}_{\rho_{14}} &= \{\} \\
\mathcal{MA}_{\rho_{14}} &= \{\} \\
\mathcal{E}_{\rho_{14}} &= \{ErrorHandling, AbstractOperationMonitor\} \\
\mathcal{ME}_{\rho_{14}} &= \{(ErrorHandling, 1), (AbstractOperationMonitor, 1)\}
\end{aligned}
$$

```
1   Object around(final Object controller, final String
      methodName) :
```

```
2       methodNameControllerExec(controller, methodName) &&
            monitorEnabled() {
3     RequestContext rc = new OperationRequestContext() {
4       public Object doExecute() {
5         return proceed(controller, methodName);
6       }
7       protected Object getKey() {
8         return methodName;
9       }
10    };
11    return rc.execute();
12  }
```

D.3.15 ρ_{15} - around(Object): methodCon...

$$
\begin{aligned}
\rho_{15} &= around(Object) : methodCon\ldots \\
locc(\rho_{15}) &= 11 \\
wom(\rho_{15}) &= 2 \\
cda(\rho_{15}) &= 0 \\
cae(\rho_{15}) &= 2 \\
\mathcal{A}_{\rho_{15}} &= \{\} \\
\mathcal{MA}_{\rho_{15}} &= \{\} \\
\mathcal{E}_{\rho_{15}} &= \{ErrorHandling, AbstractOperationMonitor\} \\
\mathcal{ME}_{\rho_{15}} &= \{(ErrorHandling, 1), (AbstractOperationMonitor, 1)\}
\end{aligned}
$$

```
1   Object around(final Object controller, final Method method)
        :
2       methodControllerExec(controller, method) &&
            monitorEnabled() {
3     RequestContext rc = new OperationRequestContext() {
4       public Object doExecute() {
5         return proceed(controller, method);
6       }
7       protected Object getKey() {
8         return concatenatedKey(controller.getClass(), method.
              getName());
9       }
10    };
11    return rc.execute();
12  }
```

D.4 Impact Functions and Their Values

Table D.1 shows the relationship between the super-aspect, the sub-aspects and the advices. Each application of the *Pull Up Advice* refactoring pattern is represented by a

numbered λ. For example, the λ_1 application moves the ρ_1 advice (*around(DataSource)*) from the β_1 sub-aspect (*JDBCConnectionMonitor*) to the α_1 super-aspect (*AbstractResourceMonitor*).

Table D.1: Relationship between the super-aspect, the sub-aspects and the advices.

λ	α	β	ρ
λ_1	α_1	β_1	ρ_1
λ_2	α_1	β_1	ρ_2
λ_3	α_1	β_2	ρ_3
λ_4	α_1	β_2	ρ_4
λ_5	α_1	β_2	ρ_5
λ_6	α_1	β_2	ρ_6
λ_7	α_1	β_3	ρ_7
λ_8	α_1	β_3	ρ_8
λ_9	α_1	β_4	ρ_9
λ_{10}	α_2	β_5	ρ_{10}
λ_{11}	α_3	β_6	ρ_{11}
λ_{12}	α_3	β_6	ρ_{12}
λ_{13}	α_3	β_6	ρ_{13}
λ_{14}	α_3	β_6	ρ_{14}
λ_{15}	α_3	β_6	ρ_{15}

The impact function values for λ are as follows:

$$
\begin{aligned}
\lambda_1 &= \lambda(\alpha_1, \beta_1, \rho_1) \\
f(\lambda_1, \alpha_1, locc) &= locc(\rho_1) = 10 \\
f(\lambda_1, \alpha_1, nom) &= 1 + nom(\rho_1) = 2 \\
f(\lambda_1, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_1} - \mathcal{E}_{\alpha_1}|] = [0, 1] \\
f(\lambda_1, \alpha_1, cda) &= [|\mathcal{A}_{\rho_1} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_1} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_1, \beta_1, locc) &= -locc(\rho_1) = -10 \\
f(\lambda_1, \beta_1, nom) &= -1 - nom(\rho_1) = -2 \\
f(\lambda_1, \beta_1, cae) &= -(|\mathcal{ME}_{\beta_1}| - |\mathcal{ME}_{\beta_1} - \mathcal{ME}_{\rho_1}|) = 0 \\
f(\lambda_1, \beta_1, cda) &= -|\mathcal{A}_{\beta_1} - (\mathcal{MA}_{\beta_1} - \mathcal{A}_{\rho_1})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_2 &= \lambda(\alpha_1, \beta_1, \rho_2) \\
f(\lambda_2, \alpha_1, locc) &= locc(\rho_2) = 10 \\
f(\lambda_2, \alpha_1, nom) &= 1 + nom(\rho_2) = 2 \\
f(\lambda_2, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_2} - \mathcal{E}_{\alpha_1}|] = [0, 1] \\
f(\lambda_2, \alpha_1, cda) &= [|\mathcal{A}_{\rho_2} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_2} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_2, \beta_1, locc) &= -locc(\rho_2) = -10 \\
f(\lambda_2, \beta_1, nom) &= -1 - nom(\rho_2) = -2 \\
f(\lambda_2, \beta_1, cae) &= -(|\mathcal{ME}_{\beta_1}| - |\mathcal{ME}_{\beta_1} - \mathcal{ME}_{\rho_2}|) = 0 \\
f(\lambda_2, \beta_1, cda) &= -|\mathcal{A}_{\beta_1} - (\mathcal{MA}_{\beta_1} - \mathcal{A}_{\rho_2})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_3 &= \lambda(\alpha_1, \beta_2, \rho_3) \\
f(\lambda_3, \alpha_1, locc) &= locc(\rho_3) = 4 \\
f(\lambda_3, \alpha_1, nom) &= 1 + nom(\rho_3) = 1 \\
f(\lambda_3, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_3} - \mathcal{E}_{\alpha_1}|] = 0 \\
f(\lambda_3, \alpha_1, cda) &= [|\mathcal{A}_{\rho_3} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_3} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_3, \beta_2, locc) &= -locc(\rho_3) = -4 \\
f(\lambda_3, \beta_2, nom) &= -1 - nom(\rho_3) = -1 \\
f(\lambda_3, \beta_2, cae) &= -(|\mathcal{ME}_{\beta_2}| - |\mathcal{ME}_{\beta_2} - \mathcal{ME}_{\rho_3}|) = 0 \\
f(\lambda_3, \beta_2, cda) &= -|\mathcal{A}_{\beta_2} - (\mathcal{MA}_{\beta_2} - \mathcal{A}_{\rho_3})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_4 &= \lambda(\alpha_1, \beta_2, \rho_4) \\
f(\lambda_4, \alpha_1, locc) &= locc(\rho_4) = 10 \\
f(\lambda_4, \alpha_1, nom) &= 1 + nom(\rho_4) = 2 \\
f(\lambda_4, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_4} - \mathcal{E}_{\alpha_1}|] = 1 \\
f(\lambda_4, \alpha_1, cda) &= [|\mathcal{A}_{\rho_4} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_4} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_4, \beta_2, locc) &= -locc(\rho_4) = -10 \\
f(\lambda_4, \beta_2, nom) &= -1 - nom(\rho_4) = -2 \\
f(\lambda_4, \beta_2, cae) &= -(|\mathcal{ME}_{\beta_2}| - |\mathcal{ME}_{\beta_2} - \mathcal{ME}_{\rho_4}|) = 0 \\
f(\lambda_4, \beta_2, cda) &= -|\mathcal{A}_{\beta_2} - (\mathcal{MA}_{\beta_2} - \mathcal{A}_{\rho_4})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_5 &= \lambda(\alpha_1, \beta_2, \rho_5) \\
f(\lambda_5, \alpha_1, locc) &= locc(\rho_5) = 6 \\
f(\lambda_5, \alpha_1, nom) &= 1 + nom(\rho_5) = 1 \\
f(\lambda_5, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_5} - \mathcal{E}_{\alpha_1}|] = 0 \\
f(\lambda_5, \alpha_1, cda) &= [|\mathcal{A}_{\rho_5} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_5} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_5, \beta_2, locc) &= -locc(\rho_5) = -6 \\
f(\lambda_5, \beta_2, nom) &= -1 - nom(\rho_5) = -1 \\
f(\lambda_5, \beta_2, cae) &= -(|\mathcal{ME}_{\beta_2}| - |\mathcal{ME}_{\beta_2} - \mathcal{ME}_{\rho_5}|) = 0 \\
f(\lambda_5, \beta_2, cda) &= -|\mathcal{A}_{\beta_2} - (\mathcal{MA}_{\beta_2} - \mathcal{A}_{\rho_5})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_6 &= \lambda(\alpha_1, \beta_2, \rho_6) \\
f(\lambda_6, \alpha_1, locc) &= locc(\rho_6) = 11 \\
f(\lambda_6, \alpha_1, nom) &= 1 + nom(\rho_6) = 3 \\
f(\lambda_6, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_6} - \mathcal{E}_{\alpha_1}|] = [0, 1] \\
f(\lambda_6, \alpha_1, cda) &= [|\mathcal{A}_{\rho_6} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_6} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_6, \beta_2, locc) &= -locc(\rho_6) = -11 \\
f(\lambda_6, \beta_2, nom) &= -1 - nom(\rho_6) = -3 \\
f(\lambda_6, \beta_2, cae) &= -(|\mathcal{ME}_{\beta_2}| - |\mathcal{ME}_{\beta_2} - \mathcal{ME}_{\rho_6}|) = 0 \\
f(\lambda_6, \beta_2, cda) &= -|\mathcal{A}_{\beta_2} - (\mathcal{MA}_{\beta_2} - \mathcal{A}_{\rho_6})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_7 &= \lambda(\alpha_1, \beta_3, \rho_7) \\
f(\lambda_7, \alpha_1, locc) &= locc(\rho_7) = 13 \\
f(\lambda_7, \alpha_1, nom) &= 1 + nom(\rho_7) = 3 \\
f(\lambda_7, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_7} - \mathcal{E}_{\alpha_1}|] = [0, 1] \\
f(\lambda_7, \alpha_1, cda) &= [|\mathcal{A}_{\rho_7} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_7} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_7, \beta_3, locc) &= -locc(\rho_7) = -13 \\
f(\lambda_7, \beta_3, nom) &= -1 - nom(\rho_7) = -3 \\
f(\lambda_7, \beta_3, cae) &= -(|\mathcal{ME}_{\beta_3}| - |\mathcal{ME}_{\beta_3} - \mathcal{ME}_{\rho_7}|) = -1 \\
f(\lambda_7, \beta_3, cda) &= -|\mathcal{A}_{\beta_3} - (\mathcal{MA}_{\beta_3} - \mathcal{A}_{\rho_7})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_8 &= \lambda(\alpha_1, \beta_3, \rho_8) \\
f(\lambda_8, \alpha_1, locc) &= locc(\rho_8) = 13 \\
f(\lambda_8, \alpha_1, nom) &= 1 + nom(\rho_8) = 3 \\
f(\lambda_8, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_8} - \mathcal{E}_{\alpha_1}|] = [0, 1] \\
f(\lambda_8, \alpha_1, cda) &= [|\mathcal{A}_{\rho_8} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_8} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_8, \beta_3, locc) &= -locc(\rho_8) = -13 \\
f(\lambda_8, \beta_3, nom) &= -1 - nom(\rho_8) = -3 \\
f(\lambda_8, \beta_3, cae) &= -(|\mathcal{ME}_{\beta_3}| - |\mathcal{ME}_{\beta_3} - \mathcal{ME}_{\rho_8}|) = -1 \\
f(\lambda_8, \beta_3, cda) &= -|\mathcal{A}_{\beta_3} - (\mathcal{MA}_{\beta_3} - \mathcal{A}_{\rho_8})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_9 &= \lambda(\alpha_1, \beta_4, \rho_9) \\
f(\lambda_9, \alpha_1, locc) &= locc(\rho_9) = 11 \\
f(\lambda_9, \alpha_1, nom) &= 1 + nom(\rho_9) = 3 \\
f(\lambda_9, \alpha_1, cae) &= [0, |\mathcal{E}_{\rho_9} - \mathcal{E}_{\alpha_1}|] = [0, 1] \\
f(\lambda_9, \alpha_1, cda) &= [|\mathcal{A}_{\rho_9} - \mathcal{A}_{\alpha_1}|, |\mathcal{A}_{\rho_9} - \mathcal{A}_{\alpha_1}| + 1] = [0, 1] \\
f(\lambda_9, \beta_4, locc) &= -locc(\rho_9) = -11 \\
f(\lambda_9, \beta_4, nom) &= -1 - nom(\rho_9) = -3 \\
f(\lambda_9, \beta_4, cae) &= -(|\mathcal{ME}_{\beta_4}| - |\mathcal{ME}_{\beta_4} - \mathcal{ME}_{\rho_9}|) = -2 \\
f(\lambda_9, \beta_4, cda) &= -|\mathcal{A}_{\beta_4} - (\mathcal{MA}_{\beta_4} - \mathcal{A}_{\rho_9})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{10} &= \lambda(\alpha_2, \beta_5, \rho_{10}) \\
f(\lambda_{10}, \alpha_2, locc) &= locc(\rho_{10}) = 17 \\
f(\lambda_{10}, \alpha_2, nom) &= 1 + nom(\rho_{10}) = 3 \\
f(\lambda_{10}, \alpha_2, cae) &= [0, |\mathcal{E}_{\rho_{10}} - \mathcal{E}_{\alpha_2}|] = [0, 1] \\
f(\lambda_{10}, \alpha_2, cda) &= [|\mathcal{A}_{\rho_{10}} - \mathcal{A}_{\alpha_2}|, |\mathcal{A}_{\rho_{10}} - \mathcal{A}_{\alpha_2}| + 1] = [0, 1] \\
f(\lambda_{10}, \beta_5, locc) &= -locc(\rho_{10}) = -17 \\
f(\lambda_{10}, \beta_5, nom) &= -1 - nom(\rho_{10}) = -3 \\
f(\lambda_{10}, \beta_5, cae) &= -(|\mathcal{ME}_{\beta_5}| - |\mathcal{ME}_{\beta_5} - \mathcal{ME}_{\rho_{10}}|) = -2 \\
f(\lambda_{10}, \beta_5, cda) &= -|\mathcal{A}_{\beta_5} - (\mathcal{MA}_{\beta_5} - \mathcal{A}_{\rho_{10}})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{11} &= \lambda(\alpha_3, \beta_6, \rho_{11}) \\
f(\lambda_{11}, \alpha_3, locc) &= locc(\rho_{11}) = 3 \\
f(\lambda_{11}, \alpha_3, nom) &= 1 + nom(\rho_{11}) = 1 \\
f(\lambda_{11}, \alpha_3, cae) &= [0, |\mathcal{E}_{\rho_{11}} - \mathcal{E}_{\alpha_3}|] = 0 \\
f(\lambda_{11}, \alpha_3, cda) &= [|\mathcal{A}_{\rho_{11}} - \mathcal{A}_{\alpha_3}|, |\mathcal{A}_{\rho_{11}} - \mathcal{A}_{\alpha_3}| + 1] = [1, 2] \\
f(\lambda_{11}, \beta_6, locc) &= -locc(\rho_{11}) = -3 \\
f(\lambda_{11}, \beta_6, nom) &= -1 - nom(\rho_{11}) = -1 \\
f(\lambda_{11}, \beta_6, cae) &= -(|\mathcal{ME}_{\beta_6}| - |\mathcal{ME}_{\beta_6} - \mathcal{ME}_{\rho_{11}}|) = 0 \\
f(\lambda_{11}, \beta_6, cda) &= -|\mathcal{A}_{\beta_6} - (\mathcal{MA}_{\beta_6} - \mathcal{A}_{\rho_{11}})| = -1
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{12} &= \lambda(\alpha_3, \beta_6, \rho_{12}) \\
f(\lambda_{12}, \alpha_3, locc) &= locc(\rho_{12}) = 11 \\
f(\lambda_{12}, \alpha_3, nom) &= 1 + nom(\rho_{12}) = 3 \\
f(\lambda_{12}, \alpha_3, cae) &= [0, |\mathcal{E}_{\rho_{12}} - \mathcal{E}_{\alpha_3}|] = 0 \\
f(\lambda_{12}, \alpha_3, cda) &= [|\mathcal{A}_{\rho_{12}} - \mathcal{A}_{\alpha_3}|, |\mathcal{A}_{\rho_{12}} - \mathcal{A}_{\alpha_3}| + 1] = [0, 1] \\
f(\lambda_{12}, \beta_6, locc) &= -locc(\rho_{12}) = -11 \\
f(\lambda_{12}, \beta_6, nom) &= -1 - nom(\rho_{12}) = -3 \\
f(\lambda_{12}, \beta_6, cae) &= -(|\mathcal{ME}_{\beta_6}| - |\mathcal{ME}_{\beta_6} - \mathcal{ME}_{\rho_{12}}|) = 0 \\
f(\lambda_{12}, \beta_6, cda) &= -|\mathcal{A}_{\beta_6} - (\mathcal{MA}_{\beta_6} - \mathcal{A}_{\rho_{12}})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{13} &= \lambda(\alpha_3, \beta_6, \rho_{13}) \\
f(\lambda_{13}, \alpha_3, locc) &= locc(\rho_{13}) = 11 \\
f(\lambda_{13}, \alpha_3, nom) &= 1 + nom(\rho_{13}) = 3 \\
f(\lambda_{13}, \alpha_3, cae) &= [0, |\mathcal{E}_{\rho_{13}} - \mathcal{E}_{\alpha_3}|] = 0 \\
f(\lambda_{13}, \alpha_3, cda) &= [|\mathcal{A}_{\rho_{13}} - \mathcal{A}_{\alpha_3}|, |\mathcal{A}_{\rho_{13}} - \mathcal{A}_{\alpha_3}| + 1] = [0, 1] \\
f(\lambda_{13}, \beta_6, locc) &= -locc(\rho_{13}) = -11 \\
f(\lambda_{13}, \beta_6, nom) &= -1 - nom(\rho_{13}) = -3 \\
f(\lambda_{13}, \beta_6, cae) &= -(|\mathcal{ME}_{\beta_6}| - |\mathcal{ME}_{\beta_6} - \mathcal{ME}_{\rho_{13}}|) = 0 \\
f(\lambda_{13}, \beta_6, cda) &= -|\mathcal{A}_{\beta_6} - (\mathcal{MA}_{\beta_6} - \mathcal{A}_{\rho_{13}})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{14} &= \lambda(\alpha_3, \beta_6, \rho_{14}) \\
f(\lambda_{14}, \alpha_3, locc) &= locc(\rho_{14}) = 11 \\
f(\lambda_{14}, \alpha_3, nom) &= 1 + nom(\rho_{14}) = 3 \\
f(\lambda_{14}, \alpha_3, cae) &= [0, |\mathcal{E}_{\rho_{14}} - \mathcal{E}_{\alpha_3}|] = 0 \\
f(\lambda_{14}, \alpha_3, cda) &= [|\mathcal{A}_{\rho_{14}} - \mathcal{A}_{\alpha_3}|, |\mathcal{A}_{\rho_{14}} - \mathcal{A}_{\alpha_3}| + 1] = [0, 1] \\
f(\lambda_{14}, \beta_6, locc) &= -locc(\rho_{14}) = -11 \\
f(\lambda_{14}, \beta_6, nom) &= -1 - nom(\rho_{14}) = -3 \\
f(\lambda_{14}, \beta_6, cae) &= -(|\mathcal{ME}_{\beta_6}| - |\mathcal{ME}_{\beta_6} - \mathcal{ME}_{\rho_{14}}|) = 0 \\
f(\lambda_{14}, \beta_6, cda) &= -|\mathcal{A}_{\beta_6} - (\mathcal{MA}_{\beta_6} - \mathcal{A}_{\rho_{14}})| = 0
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{15} &= \lambda(\alpha_3, \beta_6, \rho_{15}) \\
f(\lambda_{15}, \alpha_3, locc) &= locc(\rho_{15}) = 11 \\
f(\lambda_{15}, \alpha_3, nom) &= 1 + nom(\rho_{15}) = 3 \\
f(\lambda_{15}, \alpha_3, cae) &= [0, |\mathcal{E}_{\rho_{15}} - \mathcal{E}_{\alpha_3}|] = 0 \\
f(\lambda_{15}, \alpha_3, cda) &= [|\mathcal{A}_{\rho_{15}} - \mathcal{A}_{\alpha_3}|, |\mathcal{A}_{\rho_{15}} - \mathcal{A}_{\alpha_3}| + 1] = [0, 1] \\
f(\lambda_{15}, \beta_6, locc) &= -locc(\rho_{15}) = -11 \\
f(\lambda_{15}, \beta_6, nom) &= -1 - nom(\rho_{15}) = -3 \\
f(\lambda_{15}, \beta_6, cae) &= -(|\mathcal{ME}_{\beta_6}| - |\mathcal{ME}_{\beta_6} - \mathcal{ME}_{\rho_{15}}|) = 0 \\
f(\lambda_{15}, \beta_6, cda) &= -|\mathcal{A}_{\beta_6} - (\mathcal{MA}_{\beta_6} - \mathcal{A}_{\rho_{15}})| = 0
\end{aligned}
$$

APPENDIX E COMPUTING THE EFFECTS OF REFACTORING ON OBJECT-ORIENTED SOFTWARE

In this appendix, the definition of impact functions for class diagrams is exemplified. First, the targets for refactoring are selected from the existing metamodel elements. Next, refactoring patterns for those elements are selected and last, impact functions for a set of metrics are created to simulate the values of those metrics when the refactoring patterns are applied.

E.1 Selecting Targets.

As the example uses the UML metamodel, the first step is to select the target elements from all the UML class diagrams metamodel, that can be denoted as an $\mathcal{E}$ set:

$$\begin{aligned}\mathcal{E} \;=\; & \{Class, Classifier, Generalization, Operation, Property,\\ & StructuralFeature, AggregationKind, ValueSpecification,\\ & Type, Association, Relationship\}\end{aligned}$$

To exemplify the process, only one class from the $\mathcal{E}$ set was selected: the *Class* class. Also, two associations are being considered: the association of a *Class* instance with a *Property* and the association of a *Class* with an *Operation*. In this case, the targets are the sub-set:

$$K = \{Class, Property, Operation\}$$

E.2 Selecting Refactoring Patterns.

The more commonly available refactoring patterns in IDEs and Case tools for the target elements are:

- Class: Extract Sub-Class, Inline Class, Collapse Hierarchy, Extract Interface, Extract Super-Class, Extract Class.
- Operation: Move Method, Inline Method, Pull Up Method, Push Down Method.
- Property (Attribute): Encapsulate Field.

To exemplify the approach, a composed refactoring pattern named Extract Sub-Class is used, which uses other simpler refactoring patterns to manipulate classes, operations and attributes.

E.3 Creating the Impact Functions.

The refactoring pattern discussed is Extract Sub-Class. This pattern is a composition of New Sub-Class and a set of Push Down Attribute or Push Down Method refactoring patterns. Push down refactoring patterns can be seen as a Move refactoring pattern or a combination of a set of the application of a Copy followed by a Delete transformation.

The process starts by defining impact functions for Copy Attribute.

Copy Attribute. It can be represented as a function that receives two classes (source and destination) and one attribute as parameters and evaluates to a 2-tuple containing the modified classes (s' and d'):

$$ca(s, d, a) : Class\ Class\ Attribute \rightarrow Tuple(s' : Class, d' : Class) \tag{E.1}$$

where *s* is the source class, *d* is the destination class, *a* is the attribute to be copied and $s'(ca(s, d, a))$ and $d'(ca(s, d, a))$ returns from the 2-tuple the modified source and the modified destination respectively. Note that there is the need to define impact functions for both participating classes, as the selected metrics are class-based ones. *Impact Functions.* The impact functions for the Copy Attribute refactoring pattern can be defined as:

$$\begin{aligned} f(ca(s, d, a), s, \eta) &= \eta(s'(ca(s, d, a))) - \eta(s) \\ f(ca(s, d, a), d, \eta) &= \eta(d'(ca(s, d, a))) - \eta(d) \end{aligned}$$

As this particular refactoring pattern does not change the selected metrics, the predicate $\eta(s'(ca(s, d, a))) = \eta(s) \wedge \eta(d'(ca(s, d, a))) = \eta(d)$ is true for all the metrics. In this case, the impact functions $\forall m \in M$ are:

$$\begin{aligned} f(ca(s, d, a), s, m) &= 0 \\ f(ca(s, d, a), d, m) &= 0 \end{aligned}$$

All impact functions of Copy Attribute evaluate to zero. The same occurs with the Delete Attribute refactoring pattern.

Copy Method. The Copy Method transformation requires two classes: a source class (*s*) and a destination class (*d*). It also requires as a parameter the method to be moved (*m*): The refactoring pattern can represented by a function:

$$cm(s, d, m) : Class\ Class\ Method \rightarrow Tuple(s' : Class, d' : Class)$$

Impact Functions. This refactoring pattern only changes the wmc metric for the destination class. Therefore, the impact function for the Copy Method regarding wmc in the destination class can be defined as:

$$f(cm(s, d, m), d, wmc) = wmc(d'(cm(s, d, m))) - wmc(d)$$

As the number of operations in the destination class increases in one, $wmc($ $d'(cm(s, d, m))) = (wmc(d) + 1)$, the impact function can be simplified as follows:

$$\begin{aligned} f(cm(s, d, m), d, wmc) &= (wmc(d) + 1) - wmc(d) \\ f(cm(s, d, m), d, wmc) &= +1 \end{aligned}$$

All other impact functions evaluate to zero and to avoid repetition will not be described here. The simplification steps for the next refactoring patterns will be more concise.

Delete Method. This refactoring pattern deletes a method *m* of a class *c* and returns a modified class c':

$$dm(c, m) : Class\ Method \rightarrow Class$$

Impact Functions. It only changes the wmc metric, soh the only impact function that does not return zero is:

$$\begin{aligned} f(dm(c, m), c, wmc) &= (wmc(c) - 1) - wmc(c) \\ f(dm(c, m), c, wmc) &= -1 \end{aligned}$$

Move Method. The Move Method refactoring pattern can be seen as the application of a Copy Method and a Delete Method transformations and can be represented as function that receives two classes (`s` and `d`, representing the source and destination) and a method (`m`) as a parameter and returns two modified classes (s' and d') representing the modified source and destination:

$$mm(s, d, m) : Class\ Class\ Method \rightarrow Tuple(s' : Class, d' : Class) \tag{E.2}$$

Impact Functions. The impact function for this refactoring, for a source class *s*, a destination class *d* and a method *m* is:

$$\begin{aligned} f(mm(s, d, m), s, \eta) &= f(cm(s, d, m), s, \eta) + f(dm(c, m), s, \eta) \\ f(mm(s, d, m), d, \eta) &= f(cm(s, d, m), d, \eta) \end{aligned}$$

New Subclass. Considering a superclass *c* and a new class *nc*, the New Sub-Class refactoring pattern can be defined as:

$$nsc(c, nc) \ : \ Class\ Class$$

Impact Functions. The impact functions are:

$$\begin{aligned} f(nsc(c, nc), nc, dit) &= dit(c) + 1 \\ f(nsc(c, nc), c, noc) &= +1 \end{aligned}$$

Extract Subclass. Using the impact functions for New Sub-Class, Copy Method, Copy Attribute, Delete Attribute and Delete Method, the Extract Sub-Class impact function can be defined. Consider a class *c*, a new class *nc*, a set of methods *M* and a set of attributes *A* to be moved to the subclass, a function to represent the refactoring pattern can be defined as:

$$esc(c, nc, M, A) \ : \ Class\ Class\ Set < Method >\ Set < Attribute >$$

Impact Functions. The impact functions can be defined by the composition of the impact functions computed from the application of:

- One New Sub-Class
- Several Move Method
- Several Move Attribute

This is represented by a sum of the values of the respective impact functions:

$$f(esc(c, nc, M, A), c, \eta) = f(nsc(c, nc), c, \eta) + \sum_{i=1}^{|M|} f(mm(c, n, m_i), c, \eta)$$

$$f(esc(c, nc, M, A), nc, \eta) = f(nsc(c, nc), nc, \eta) + \sum_{i=1}^{|M|} f(mm(c, n, m_i), nc, \eta)$$

The impact functions for the Move Attribute refactoring pattern were omitted as it does not change the selected metrics in $\mathcal{M}$.

APPENDIX F IMPACT FUNCTIONS FOR EXTRACT POINTCUT AND INLINE INTER-TYPE FIELD DECLARATION

This appendix defines impact functions for the Extract Pointcut and the Inline Inter-Type Field Declaration refactoring patterns for the metrics: *locc*, *nom*, *cda* and *cae*.

F.1 Extract Pointcut

Consider a λ refactoring pattern, corresponding to the application of an *Extract Pointcut* refactoring pattern to a μ module. The p expression denotes the pointcut expression being extracted and s is the name of the new pointcut:

$$\lambda = extractPointcut\ (\mu : Module, p : PointcutExpression, s : Identifier) \quad \text{(F.1)}$$

The application of this refactoring pattern creates a new pointcut named s and copies the p expression to this pointcut. The application of λ replaces the occurrences to the expression to use the name of the newly created pointcut.

F.1.1 Impact on $locc$.

The changes in the $locc$ metric value after the application of an *Extract Pointcut* refactoring pattern in any given module μ could be calculated using the following function:

$$f(\lambda, \mu, locc) = locc(p) \quad \text{(F.2)}$$

In the function used to calculate $locc$ only $locc(\mathcal{P})$ changes, so $f(\lambda, \mu, locc) = locc(\mathcal{P}') - locc(\mathcal{P})$. As $P' = P \cup p$, one can conclude that $f(\lambda, \mu, locc) = locc(p)$.

Usually, a pointcut is defined in one single line of code. However, as programming languages designers could define different constructions to represent the same abstraction mechanism and can define a multi-line construction to express pointcuts, it was opted to leave the impact function as $f(\lambda, \mu, locc) = +locc(p)$ instead of a simplified version $f(\lambda, \mu, locc) = +1$ (as is in AspectJ or CaesarJ (MEZINI; OSTERMANN, 2003), for example).

F.1.2 Impact on nom.

The nom metric value is not affected by this refactoring pattern because the number of methods, advices, inter-type method declarations and inter-type constructor declarations do not change when the transformation occurs.

F.1.3 Example.

Consider a *Debug* aspect, part of an example named *Space War* (a spaceship and asteroids game (HILSDALE; KICZALES, 2001)). This aspect is responsible for keeping and displaying debug information. In this aspect, there is a pointcut directly defined in the *after* advice (line 2). The pointcut definition can be extracted from the advice to provide a name for the affected points and to enable the reuse of the extracted expression.

```
1  aspect Debug {
2     after(Ship ship, SpaceObject obj) returning :
3     call(void Ship.handleCollision(SpaceObject))
4     && target(ship) && args(obj){
5     ...
6  ...
7  }
```

Using the *aopmetrics* (STOCHMIALEK, 2009) tool to gather the metric values before extracting the *handleCollision* pointcut, the developer gets the following values:

$$Debug = \{locc = 83, nom = 13\} \tag{F.3}$$

The resulting metric values after the application of a *Extract Pointcut* refactoring pattern can be obtained using the defined impact functions. In this case, only the $locc$ metric is modified, according to the function by increasing the $locc$ value by one unit. Using the impact function, the new $locc$ value is:

$$\begin{aligned} \mu &= Debug \\ p &= call(voidShip.handleCollision(SpaceObject)) \\ &\quad \&\& \; target(ship) \; \&\& \; args(obj) \\ \lambda &= extractPointcut(\mu, p, "collision") \\ locc'(\mu) &= locc(\mu) + f(\lambda, \mu, locc) = 83 + 1 = 84 \end{aligned}$$

The values calculated using the impact functions prior the application of the refactoring pattern are the same obtained using the *aopmetrics* tool after applying the refactoring pattern:

$$Debug = \{locc = 84, nom = 13\} \tag{F.4}$$

After the extraction, the affected points are defined in a named pointcut (line 2). The *collision* pointcut provides a semantic definition to the predicate that in the previous example was attached directly to the advice and can be reused by other advices of this aspect.

```
1  aspect Debug {
2     pointcut collision(Ship ship, SpaceObject obj):
3        call(void Ship.handleCollision(SpaceObject))
4        && target(ship) && args(obj);
5        ...
6        after(Ship ship, SpaceObject obj) returning : collision
            (ship, obj){ ... }
```

F.2 Inline Inter-Type Field Declaration

Consider an aspect α, a class β and an inter-type field declaration (ITFD) ϕ. This refactoring pattern could be defined as:

$$\lambda = inlineITFD(\alpha : Aspect, \beta : Class, \phi : ITFD) \tag{F.5}$$

F.2.1 Impact on $locc$.

Inlining an inter-type field declaration only changes the $locc$ metric values by one unit as the field no longer is declared in the aspect, but is inserted directly in the affected class:

$$f(\lambda, \alpha, locc) = -1 \tag{F.6}$$
$$f(\lambda, \beta, locc) = +1 \tag{F.7}$$

F.2.2 Impact on nom.

The nom metric value is not affected by this refactoring pattern because the number of methods, advices, inter-type method declarations and inter-type constructor declarations are not changed when the *Inline Inter-type Field Declaration* refactoring pattern is applied.

F.2.3 Impact on cda.

Let $\mathcal{ME} = (\mathcal{E}, m)$ be the multiset of all modules advised by α, where $\mathcal{E}$ is the set of advised modules and $m : \mathcal{E} \rightarrow N$ is a function from $\mathcal{E}$ to the set of natural numbers. For each $e \in \mathcal{E}$ the multiplicity of e is given by $m(e)$. Let β be the class affected by the inter type field declaration. The function that defines if the cda metric value is decreased could be seen as:

$$f(\lambda, \alpha, cda) = \begin{cases} 0 & : \quad m(\beta) > 1 \\ -1 & : \quad otherwise \end{cases} \tag{F.8}$$
$$f(\lambda, \beta, cda) = 0 \tag{F.9}$$

F.2.4 Impact on cae.

The cae metric value can change if the aspect containing the inter-type declaration no longer affects the class. Let $\mathcal{MA} = (\mathcal{A}, m)$ be the multiset of all modules that advises β, where $\mathcal{A}$ is the set of advising modules and $m : \mathcal{A} \rightarrow N$ is a function from $\mathcal{A}$ to the set of natural numbers. For each $a \in A$ the multiplicity of a is given by $m(a)$. Let β be the class affected by the inter type field declaration. The impact function for the cae metric for α and β is:

$$f(\lambda, \alpha, cae) = 0 \tag{F.10}$$
$$f(\lambda, \beta, cae) = \begin{cases} -1 & : \quad m(\beta) = 1 \\ 0 & : \quad otherwise \end{cases} \tag{F.11}$$

F.2.5 Example.

In the *spacewar* example, the *SpaceObjectPainting* (lines 1-4, an inner aspect of the *Display1* aspect) contains an inter-type field declaration (line 2) named *color* to the *Ship* class (line 5). This inter-type declaration will be inserted in the class using the *Inline Inter-Type Field Declaration* refactoring pattern.

```
1 static aspect SpaceObjectPainting {
2     private Color Ship.color;
3     ...
4 }
5 class Ship extends SpaceObject {
6     ...
7 }
```

The values of the selected metrics before the refactoring process are:

$$\begin{aligned} Display1.SpaceObjectPainting &= \{locc = 64, nom = 5, cda = 6, cae = 0\} \\ Ship &= \{locc = 193, nom = 21, cda = 0, cae = 3\} \end{aligned}$$

Evaluating the changes in both *SpaceObjectPainting* aspect and *Ship* class using the defined impact functions:

$$\begin{aligned} \alpha &= Display1.SpaceObjectPainting,\ \beta = Ship \\ \lambda &= inlineITFD(\alpha, \beta, ship) \\ locc'(\alpha) &= locc(\alpha) + f(\lambda, \alpha, locc) = 193 + 1 = 194 \\ locc'(\beta) &= locc(\beta) + f(\lambda, \beta, locc) = 64 - 1 = 63 \end{aligned}$$

To calculate the vale for cda metric, the modules that are advised by α were collected and a multiset named $\mathcal{ME}$ was created:

$$\begin{aligned} \mathcal{ME} &= \{(SpaceObject, 1), (Ship, 2), (Bullet, 1), \\ &\quad (EnergyPacket, 1), (Game, 3), (Robot, 1)\} \\ m(Ship) &= 2 \\ cda'(\alpha) &= cda(\alpha) + f(\lambda, \alpha, cda) = 6 + 0 = 6 \\ cda'(\beta) &= cda(\beta) + f(\lambda, \beta, cda) = 0 + 0 = 0 \end{aligned}$$

The cae value for α remains the same. The cae for β changes if and only if α affects β only by introducing the field. If α affects the β class by introducing other fields, methods or constructions using inter-type declarations or advising β, the value for cae remains unchanged. To calculate the value, the multiset of modules that affect β was collected:

$$\begin{aligned} \mathcal{MA} &= \{(Display1.ShipObjectPainting, 2), \\ &\quad (Display2.ShipObjectPainting, 2), (Debug, 50)\} \\ cae'(\alpha) &= cae(\alpha) + f(\lambda, \alpha, cae) = 0 + 0 = 0 \\ m(\alpha) &= 2 \\ cae'(\beta) &= cae(\beta) + f(\lambda, \beta, cae) = 4 + m(\alpha) = 3 + 0 = 3 \end{aligned}$$

The evaluated values are the same calculated using the *aopmetric* tool after the refactoring:

$$\begin{aligned} SpaceObjectPainting &= \{locc = 63, nom = 5, cda = 6, cae = 0\} \\ Ship &= \{locc = 194, nom = 21, cda = 0, cae = 3\} \end{aligned}$$

After the refactoring, the *color* field (line 5) is declared in the *Ship* class (line 4). The modifier was changed to public, so the aspect still has access to that field. A refactoring

pattern that can be applied to use methods to access the value of *color* is the *Encapsulate Field* refactoring pattern, which replaces all the readings and writings to method calls (methods *getColor* and *setColor*)

```
1 static aspect SpaceObjectPainting {
2     ...
3 }
4 class Ship extends SpaceObject {
5     public Color color;
6         ...
7 }
```