Concern-Oriented Heuristic Assessment of Design Stability

Eduardo Magno Lages Figueiredo

Submitted for the Degree of PhD

Supervised by Prof. Jon Whittle and
Dr. Alessandro Garcia

23 October 2009

Computing Department
Lancaster University
UK

But thanks be to God!

He gives us the victory through our Lord Jesus Christ.

1 Corinthians 15:57

Acknowledgements

First and foremost, I would like to specially thank my supervisors Professor Jon Whittle and Dr. Alessandro Garcia. In particular, I want to express my appreciation to Jon who believed in my strength and took the role of my supervisor in the last year of my PhD research. Very special thanks to my supervisor and friend Alessandro for giving me freedom to shape my research path and for guiding me with his extensive knowledge and enthusiasm. I don't have words to express my sincere gratitude to Alessandro, but I consider myself fortunate to have worked with him in the last five years; two of them in my MSc course. I am also grateful to my former supervisor, Professor Carlos Lucena, who supported me since my application to have the PhD degree abroad. He helped me not only in the very beginning of my research career but also in all my way until I get my degree.

I am also thankful to the members of my examination panel, Dr. Michel Wermelinger and Dr. Pete Sawyer, who have generously committed their time and expertise to review my thesis. In addition to Michel and Pete, I would like to thanks Alberto Sardinha, Bruno Silva, Claudio Sant'Anna, Fabiano Ferrari, Leonardo Tizzei, and Nelio Cacho for their comments on an earlier draft of this thesis. They have contributed a lot to this work by providing precious comments and suggestions that greatly improved the quality of my thesis.

I am lucky to have the opportunity for collaboration with many brilliant researchers who contributed a lot to this thesis. Individually, I want to thank: (i) Claudio Sant'Anna, for all the fruitful discussions and for the time we spent together measuring and analysing measurement results; (ii) Bruno Silva for the insightful discussions about the crosscutting patterns, their formalisation, and methods of identifying them; (iii) Nelio Cacho for all the shared knowledge and experience on the empirical studies and for being a great friend; (iv) Jose Conejero for the time we spent together working on measurement of software architectures and requirements, and (v) Philip Greenwood for the good pieces of advice about my research.

I also owe a great deal of gratitude to Alessandro, Bruno, and Jon who made their students available to serve as participants in the empirical experiment described in Chapter 6 of this thesis. I would also like to thank their students for taking the

experiment seriously. Thanks to Martin Robillard, Gail Murph, and Marc Eaddy, for providing me with the source code of their tools, ConcernMapper and ConcernTagger, which made easier my implementation work.

My colleagues in the Computing Department at Lancaster University were responsible for creating a pleasant and stimulating working environment during the three years of my research. Along with the Computing Department folks, I had the pleasure of working with other great researchers in the last three years. It is hard to name them all but I will try to list some extraordinary folks (sorry if I missed someone). My thanks to Afonso Pimentel, Alessandro Marchetto, Ana Luisa Medeiros, Ana Moreira, Awais Rashid, Cecilia Rubira, Cidiane Lobato, Daltro Nunes, Elena Jurado, Fabiano Ferrari, Fernando Castor, Francisco Dantas, Fred Lopes, Glauco Carneiro, Hitalo Silva, Ismenia Galvao, Klessis Dias, Jefferson Amorim, Jose Maldonado, Juan Hernandez, Julio Leite, Leandra Dias, Liana Soares, Lidia Fuentes, Lyrene Fernandes, Marcos Dosea, Marcos Silva, Mario Monteiro, Mehmet Aksit, Monica Pinto, Otavio Lemos, Paulo Borba, Paulo Masiero, Pim van den Broek, Rachel Burrows, Rita Ribeiro, Safoora Khan, Sergio Soares, Steffen Zschaler, Thais Batista, Thiago Bartolomei, Uira Kulesza, Vander Alves, and Walter Cazzola. Surely, these guys contributed to my research in many different ways.

I also want to express my sincere thanks to Yvonne, Chris Needham, Cath, Gillian, Liz, Cheryl, and Aimee for their kind and effective way of solving all administrative issues.

My doctoral studies have been financially supported by CAPES - Brazil. I also received funding from the AMPLE project, Computing Department, Faculty of Science and Technology, Willian Ritchie Funding, and Pe Leonel Franca Foundation, which made possible my regular participation in a number of conferences. The funders of this work have my gratitude.

Last, but not least, I would like to warmly thank my family in Brazil. Thanks to my wonderful parents, Sonia and Jose, to my brothers, Angelo and Marcelo, to my sister Rosane, and to my sweet goddaughter Marina. Despite being so far away from them for such a long time, I could not finish up this thesis without their help and love.

Abstract

Software systems are always changing to address new stakeholders' concerns. Design modularity improves the stability of software by decoupling design concerns that are likely to change so that they can be maintained independently. Despite the efforts of modern programming languages, some concerns cannot be well modularised in the system design and implementation. These concerns, called crosscutting concerns, are often blamed to hinder design modularity and stability. However, recent studies have pointed out that crosscutting concerns are not always harmful to design stability. Hence, software maintainers would benefit from well documented patterns of crosscutting concerns and a better understanding about their actual impact on design stability. This document presents a catalogue of *crosscutting patterns* empirically observed in several software systems. These patterns are described and classified based on an intuitive vocabulary that facilitates their recognition by software engineers. Crosscutting patterns are detected by a heuristic assessment technique composed of metrics and heuristic rules. We also proposed a formalism to define the crosscutting patterns and their means of detection. This formalism includes a measurement framework and a meta-model of the concern realisation. The heuristic assessment technique is supported by a prototype tool which automates the detection of crosscutting patterns in software designs. The accuracy of the heuristic technique is assessed through their application to seven systems. Then, we select three of these systems to empirically evaluate the correlation of crosscutting patterns and design stability. The outcomes of our exploratory evaluation indicate that: (i) a high number of crosscutting patterns in a module leads to module instabilities, and (ii) certain specific categories of crosscutting patterns seem to be good indicators of design instabilities.

Disclaimer and List of Publications Resulting from Work on this Thesis

This dissertation has not been submitted in support of an application for another degree at this or any other university. It is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated.

Excerpts of this thesis have been published in journal, conference and workshop papers listed below. The contributions to research and writing for each paper are noted, except for workshop papers. The list is sorted so that papers which I had major contribution come first. Where available, acceptance rates of conferences and workshops are indicated.

Journal and Conference

- "Crosscutting Patterns and Design Stability: An Exploratory Analysis", E. Figueiredo, B. Silva, C. Sant'Anna, A. Garcia, J. Whittle, and D. Nunes. *In Proceedings of the 17th International Conference on Program Comprehension (ICPC)*. Vancouver, Canada, 17-19 May 2009.

  Acceptance Rate: 27%

  Research Contribution: high, Paper writing contribution: high (main author)

- "Applying and Evaluating Concern-Sensitive Design Heuristics", E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena. *In Proceedings of the 23rd Brazilian Symposium on Software Engineering (SBES)*. Fortaleza, Brazil, 5-9 October 2009.

  Acceptance Rate: 19%

  Research Contribution: high, Paper writing contribution: high (main author)

- “ConcernMorph: Metrics-based Detection of Crosscutting Patterns”, E. Figueiredo, J. Whittle, and A. Garcia. *In Proceedings of the 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, demo session. Amsterdam, The Netherlands, 24-28 August 2009.

  Acceptance Rate: 32%

  Research Contribution: high, Paper writing contribution: high (main author)

- “Detecting Architecture Instabilities with Concern Traces: An Exploratory Study”, E. Figueiredo, I. Galvao, S. Khan, A. Garcia, C. Sant'Anna, A. Pimentel, A. Medeiros, L. Fernandes, T. Batista, R. Ribeiro, P. van den Broek, M. Aksit, S. Zschaler, and A. Moreira. *In Proceedings of the Joint 8th Working IEEE/IFIP Conference on Software Architecture (WICSA) and the 3rd European Conference on Software Architecture (ECSA)*. Cambridge, UK, 14-17 September 2009.

  Acceptance Rate: 39%

  Research Contribution: high, Paper writing contribution: high (main author)

- “Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability”, E. Figueiredo, N. Cacho, C. Sant’Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas. *In Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pp. 261-270. Leipzig, Germany, 10-18 May 2008.

  Acceptance Rate: 15%

  Research Contribution: high, Paper writing contribution: high (main author)

- “On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework”, E. Figueiredo, C. Sant’Anna, A. Garcia, T. Bartolomei, W. Cazzola, and A. Marchetto. *In Proceedings of the 12th*

European Conference on Software Maintenance and Reengineering (CSMR), pp. 183-192. Athens, Greece, 1-4 April 2008.

Acceptance Rate: 28%

Research Contribution: high, Paper writing contribution: high (main author)

- "Concern-Sensitive Heuristic Assessment of Aspect-Oriented Design", E. Figueiredo and A. Garcia. *In Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP)*, Doctoral Symposium. Paphos, Cyprus, 7-11 July 2008.

  Research Contribution: high, Paper writing contribution: high (main author)

- "AJATO: an AspectJ Assessment Tool", E. Figueiredo, A. Garcia, and C. Lucena. *In Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, Demo Session. Nantes, France, 3-7 July 2006.

  Research Contribution: high, Paper writing contribution: high (main author)

- "On the Modularization and Reuse of Exception Handling with Aspects", F.C. Filho, N. Cacho, E. Figueiredo, A. Garcia, C. Rubira, J.S. Amorim, and H.O. Silva. *Software: Practice and Experience (SP&E)*, to appear, (2009).

  Research Contribution: medium, Paper writing contribution: medium

- "Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics", B. Silva, E. Figueiredo, A. Garcia, and D. Nunes. *Electronic Notes Theoretical Computer Science*, 233 (2009): 105-125.

  Research Contribution: medium, Paper writing contribution: medium

- "On the Support and Application of Macro-Refactorings for Crosscutting Concerns", B. Silva, E. Figueiredo, A. Garcia, and D. Nunes. *In Proceedings of the 3rd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS).* Natal, Brazil, 09-11 September 2009.

  Research Contribution: medium, Paper writing contribution: medium

- “Early Crosscutting Metrics as Predictors of Software Instability”, J. Conejero, E. Figueiredo, A. Garcia, J. Hernandez, and E. Jurado. *In Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS)*. Zurich, Switzerland, 29 June - 3 July 2009.

  Acceptance Rate: 25%

  Research Contribution: medium, Paper writing contribution: medium

- “EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming”, N. Cacho, F. Filho, A. Garcia, and E. Figueiredo. *In Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 72-83. Brussels, Belgium, 31 March - 4 April 2008.

  Acceptance Rate: 21%

  Research Contribution: medium, Paper writing contribution: low

- “On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study”, P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant’Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. *In Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, pp. 176-200. Berlin, 30 July - 03 August 2007.

  Acceptance Rate: 16%

  Research Contribution: medium, Paper writing contribution: medium

- “On the Modularity of Software Architectures: A Concern-Driven Measurement Framework”, C. Sant’Anna, E. Figueiredo, A. Garcia, and C. Lucena. *In Proceedings of the 1st European Conference on Software Architecture (ECSA)*, pp. 207-224. Madrid, Spain, 24-26 September 2007.

  Acceptance Rate: 20%

  Research Contribution: medium, Paper writing contribution: medium

- “Modularizing Design Patterns with Aspects: A Quantitative Study,” A. Garcia, C. Sant’Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa, *Transactions on Aspect-Oriented Software Development I* (2006): 36-74.

  Research Contribution: medium, Paper writing contribution: low

Workshops

- “Directives for Concern-Driven Code Refactorings”, B. Silva, E. Figueiredo, A. Garcia, and D. Nunes. *3rd Latin American Workshop on Aspect-Oriented Software Development (LA-WASP) co-located with SBES’09*. Fortaleza, Brazil. 04-05 October 2009.

- “Representing Architectural Aspects with a Symmetric Approach”, A. Garcia, E. Figueiredo, C. Sant’Anna, M. Pinto, and L. Fuentes. *Workshop on Early Aspects (EA) co-located with AOSD’09*. Charlottesville, Virginia, USA. 3 March 2009.

- “Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics”, B. Silva, E. Figueiredo, A. Garcia, and D. Nunes. *2nd International Workshop on Software Quality and Maintainability (SQM) co-located with CSMR’08*. Athens, Greece. 1 April 2008.

- “On the Modularity Assessment of Software Architectures: Do my architectural concerns count?”, C. Sant’Anna, E. Figueiredo, A. Garcia, and C. Lucena. *1st Workshop on Aspects in Architectural Description (AARCH) co-located with AOSD’07*. Vancouver, Canada. 12 March 2007.

- “On the Contributions of an End-to-End AOSD Testbed”, P. Greenwood, A. Garcia, A. Rashid, E. Figueiredo, C. Sant’Anna, N. Cacho, A. Sampaio, S. Soares, P. Borba, M. Dosea, R. Ramos, U. Kulesza, L. Fernandes, T. Bartolomei, M. Pinto, L. Fuentes, N. Gamez, A. Moreira, J. Araujo, T. Batista, A. Medeiros, F. Dantas, J. Wloka, C. Chavez, R. France, and I. Brito. *Workshop in Aspect-Oriented Requirements Engineering and Architecture Design (Early Aspects) co-located with ICSE’07*. Minneapolis, USA. 21 May 2007.

- “Towards a Unified Coupling Framework for Measuring Aspect-Oriented Programs”, T. Bartolomei, A. Garcia, C. Sant’Anna, and E. Figueiredo. *3rd International Workshop on Software Quality Assurance (SOQUA)*, pp. 46-53, co-located with FSE’06. Portland, USA. 6 November 2006.

Table of Contents

List of Figures

List of Tables

Glossary of Metrics

ATFD	Access to Foreign Data	**20**
CC	Changing Class	**26**
CDA	Crosscutting Degree of an Aspect	**18**
CDC	Concern Diffusion over Components	**15**
CDO	Concern Diffusion over Operations	**15**
CDLOC	Concern Diffusions over Lines of Code	**15**
CM	Changing Method	**26**
CSC	Concern-Sensitive Coupling	**45**
CSCC	Concern-Sensitive Component Cohesion	**46**
FCD	Feature Crosscutting Degree	**18**
ICSC	Intra-Component Concern-Sensitive Coupling	**46**
LCC	Lack of Concern-based Cohesion	**46**
LOC	Lines of Code	**17**
LOCC	Lines of Concern Code	**18**
NOA	Number of Attributes	**82**
NC	Number of Components	**82**
NOCA	Number of Concern Attributes	**18**
NOCO	Number of Concern Operations	**18**
NOF	Number of Features	**18**
NOO	Number of Operations	**82**
TC	Number of Tangled Components	**97**
TCC	Tight Class Cohesion	**20**
TO	Number of Tangled Operations	**97**
WMC	Weighted Method per Class	**20**

1. Introduction

Widely used software systems are never complete due to, for instance, the volatility of stakeholders' requirements [99, 125]. Software development is therefore incremental to keep pace with evolving requirements and new implementation technologies. In fact, maintaining a software system is the most expensive activity in the software development lifecycle [138]. Some recent industrial studies have demonstrated that around 50% of object-oriented design and code may be changed between two releases [11]. Moreover, more than 60% of change requests are accepted and implemented [98]. To reduce the overall cost of maintenance activities, the software system should be designed to mitigate the propagation of design changes. In other words, developers aim to achieve a stable design. *Design stability* is defined as the quality attribute indicating the resistance to the potential change propagation which a program developed from the design would have when it is modified [147]. Moreover, an *unstable module* is characterised by an excessive number of changes.

To support design stability, the principle of software modularity is applied to decouple decisions that are likely to change so that they can be maintained independently [123, 142]. Therefore, a modular design is expected to be stable since changes do not easily propagate from one module to another [4]. Another key benefit of modular design is the possibility of studying one system concern at a time [123]. A *concern* is any consideration that can impact on the design and maintenance of program modules [129]. According to this general definition, concerns can be, for instance, requirements, design patterns [64], or implementation mechanisms such as Caching. In a program aiming to sort an array of integers, an example of a concern could be Sorting. This concern is realised by the program elements responsible for sorting the array. Hence, it includes details such as the adopted sorting algorithm (e.g., quick sort or bubble sort). Modularity of concerns is important to mitigate the propagation of changes [84, 123]. In fact, developers have to naturally reason about concerns in maintenance activities because change requests are usually described in terms of the stakeholders' concerns, instead of the system modules. In this research work we consider the modularity of concerns to assess design stability.

This chapter briefly motivates and outlines this doctoral dissertation. Section 1.1 defines the problem we aim to address. Section 1.2 explains some limitations of the related work. Section 1.3 summarises our hypotheses and research questions. Section 1.4 presents the novel contributions of the proposed solution. Finally, Section 1.5 describes how the other chapters of this document are organised.

1.1 Problem Definition

Modern decomposition techniques aim to support modular programming by providing various abstractions and mechanisms. Despite these efforts, some concerns – called *crosscutting concerns* [88] – are scattered over the modules of the system. A crosscutting concern is a concern which cannot be easily encapsulated using mechanisms and abstractions of the selected decomposition technique [145]. Even when developers rely on good design practices, certain concerns are still crosscutting [3]. Recent studies have confirmed that some properties of crosscutting concerns lead to design modularity flaws and therefore hinder software stability [44, 73]. For instance, concern scattering tends to both destabilise software modularity properties and facilitate the propagation of changes according to empirical assessments [44, 119]. However, there is not knowledge if certain recurring forms of crosscutting concerns are always harmful to design stability [109].

The realisation of crosscutting concerns affects modularity units of a design in significantly different ways. For instance, a concern can be realised in a design by a few elements in a specific inheritance tree or it can be scattered across the entire system. Hence, designers should be given information not only on whether a concern is crosscutting or not, but also on the nature of the crosscutting. This information is important, for instance, to avoid expending effort in trying to modularise a crosscutting concern which is, in fact, not harmful to the stability of a design. Unfortunately, few works [79, 109] currently exist that focus on the identification and classification of different patterns of crosscutting concerns. More fundamentally, to the best of our knowledge, there is no empirical investigation so far on what specific patterns of crosscutting concerns are likely to deteriorate design stability.

In fact, the achievement of stable designs is far from trivial even with the use of contemporary decomposition techniques, such as aspect-oriented programming [88], to separate crosscutting concerns. Aspect-oriented programming (AOP) aims at the explicit isolation of crosscutting concerns [62, 94]. Therefore, software engineers need assessment techniques to support them identifying sources of design instability related to the inadequate modularisation of concerns. As a result, some metrics which indicate the potential ripple effects have been proposed for design stability [49, 147]. *Ripple effects* occur when changes made to a design module propagate to other parts of the design. There are also a number of metrics to quantify design modularity properties of object-oriented [29, 100] and aspect-oriented designs [152-154]. However, these metrics do not explicitly consider design concerns in their measurement processes. Without quantifying concern-specific properties, it is hard to assess the actual impact of crosscutting concerns on design stability.

More recently, new metrics have been defined to quantify properties of concerns [41, 46, 134, 151]. These metrics, called *concern metrics*, capture information about concerns realised by one or more modules. Their definitions allow the measurement of both object-oriented and aspect-oriented designs. However, the area of concern measurement is still incipient and suffers from not relying on standard terminology and formalisation. Many concern metrics are expressed in an ambiguous manner which limits their use. This issue makes concern metrics difficult to understand and use in practice [57]. In summary, there is a pressing need for (i) proper documentation of the common patterns of crosscutting concerns, (ii) mechanisms for detecting and quantifying these patterns of crosscutting concerns in software designs, and (iii) empirical studies which investigate the correlation of specific patterns of crosscutting concerns and design stability.

1.2 The State of the Art in Concern Documentation

Some recent work [41, 79, 108, 109] has defined ways to classify crosscutting concerns. For instance, Ducasse, Girba, and Kuhn [41] proposed a generic technique, called Distribution Map, to visualise and analyse the concerns of a system. Based on this technique, they identified four *patterns of concerns*: Well-Encapsulated, Crosscutting, Black Sheep, and Octopus. The authors also defined four metrics, namely Size, Touch, Spread, and Focus, to support the identification of these patterns of concerns. Although a preliminary concern classification and its means of automation were defined by Ducasse, Girba, and Kuhn [41], the proposed set is limited to only four patterns of concern. Moreover, the first two concern patterns, Well-Encapsulated and Crosscutting, are naturally part of the basic definition of a concern [88]. Therefore, only Black Sheep and Octopus can be actually considered original contributions towards the classification of crosscutting concerns. We discuss these two crosscutting concern patterns and eleven novel ones in Chapter 4.

There are also some studies [79, 108, 109] on the definition of refactorings for modularising specific types of crosscutting concerns. For instance, Hannemann, Murphy, and Kiczales [79] presented a technique for refactoring crosscutting concerns focusing on their *roles*. A concern role captures the general structure of the crosscutting concern realisation. Similarly, Marin, Moonen, and Deursen [108, 109] defined refactoring based on *crosscutting concern sorts*. Compared to the work from Ducasse, Girba, and Kuhn [41], both roles and concern sorts can be seen as preliminary and more specific types of concerns. However, the descriptions of roles and concern sorts (i) are not abstract and generic as required by the patterns of concerns and (ii) cannot be applied to either design artefacts or beyond the context of specific programming mechanisms. For instance, apart from design patterns, names of roles are based on a very restricted set of application-specific concerns, such as Currency Control [79]. Similarly, concern sorts are usually tied to implementation-specific constructs of programming languages. For example, the Declare Throws Clause and Exception Propagation concerns [108] (which are mapped to refactorings with the same names) are related to exception handling mechanisms commonly available in aspect-oriented languages [87].

As observed by Ducasse, Girba, and Kuhn [41], metrics and their associated heuristic interpretations can be used to detect patterns of crosscutting concerns – or *crosscutting patterns* from now on. In fact, this applicability of metrics is natural since they are traditionally the pivotal mechanism for assessing design quality attributes [15, 29, 95, 111]. Recently, there has been a growing body of relevant work in the literature [44, 46, 129] that analyses properties of a concern, such as scattering and tangling. Following these works, metrics have also been defined to quantify concern properties [46, 133]. However, there is a lack of more elaborated assessment techniques, such as *detection strategies* [95, 111], relying on concern abstraction. A detection strategy is a logical condition composed of metrics which captures deviations from good design principles [111]. Although there is increasing awareness of the importance of concern properties in design analysis, to the best of our knowledge, existing detection strategies do not take concern properties into consideration. Moreover, there is no use of detection strategies to support the identification of crosscutting patterns. More fundamentally, there is no systematic study that investigates whether this category of detection strategies enhances the process of identifying sources of design instabilities.

1.3 Hypotheses and Research Questions

Based on the previously described problems and on the limitations of the related work, we define the following hypothesis:

H I. *Some specific crosscutting patterns are more harmful to design stability than others.*

The null hypothesis is:

H 0. *All crosscutting patterns are equally harmful to design stability.*

Based on these hypotheses, we define two research questions as follows:

- *Does a high number of crosscutting patterns impact positively, negatively or have no impact on design stability?*
- *If crosscutting patterns impact on design stability, which ones are better indicators of software stability?*

To test the hypotheses and answer the research questions, we first documented recurring crosscutting patterns. Then, we defined a tool-supported heuristic technique which allows us to detect crosscutting patterns in several systems. Finally, we empirically evaluated (i) the efficacy of the detection technique and (ii) the correlation of crosscutting patterns and design stability.

1.4 Novel Elements of the Thesis

This section briefly describes the two key novel elements of our research work, namely a catalogue of crosscutting patterns and a set of studies to understand the empirical correlation of crosscutting patterns and design stability. A heuristic technique relying on metrics is proposed for detecting crosscutting patterns. We also formalise not only the crosscutting patterns but also the metrics used by the heuristic detection technique. These elements of the proposed approach are briefly described as follows.

Identification and formalisation of 13 crosscutting patterns [52]. These crosscutting patterns represent recurring ways that crosscutting concerns can manifest themselves in systems. They have been identified based on our experience on analysing concerns in several software systems[1]. We first describe and classify the crosscutting patterns using an intuitive vocabulary to facilitate their recognition by software engineers. Then, we rely on a formalism for analysis of concern properties to formalise each crosscutting pattern. Moreover, to automatically detect instances of crosscutting patterns we propose a heuristic technique based on metrics. Therefore, in addition to the definitions of the crosscutting patterns themselves, secondary contributions of this research work are:

- *Formalism for analysis of concern properties*. The proposed formalism relies on set theory and defines a terminology that is, as much as possible, agnostic to language and paradigm peculiarities. It includes a meta-model which defines possible relations of concerns and the system's structure. Due to its generality, the formalism can be instantiated for different modelling and programming languages.

- *A conceptual measurement framework [57]*. Relying on the proposed formalism, this conceptual framework defines a set of criteria to define meaningful and well-founded metrics. The framework also plays a pivotal role by supporting the comparison and selection of existing *concern metrics*. Concern metrics have been recently defined to support quantitative analysis of concern's modularity properties [46, 57]. These metrics, in turn, are the basic mechanisms for analysis of concern properties, such as scattering and tangling. For instance, concern metrics are the elementary parts of a detection strategy [111].

- *A heuristic technique [55] and a supporting tool [53] to detect crosscutting patterns*. We define a heuristic technique by extending complementary sets of analyses, such as detection strategies and graph-search algorithms, with the notion of concerns. This technique relies on a suite of metrics instantiated according to our conceptual measurement framework. The tool, called

[1] The following software systems helped us in the definition of the crosscutting patterns: MobileMedia [52] [54], Health Watcher [73] [137], a Design Pattern library [80], OpenOrb Middleware [22], AJATO [1] [58], Eclipse CVS Plugin [22] [36], and Portalware [68] [120]. We discuss these systems later in this document.

ConcernMorph, supports automatic identification of crosscutting patterns based on the implemented metrics and heuristic algorithms.

A controlled experiment and a set of empirical studies [52, 54, 56]. The main purpose of these evaluation studies is to investigate the impact of crosscutting patterns on design stability. However, they also include four preliminary evaluation steps to achieve this main goal. First, we perform a theoretical evaluation of the proposed formalism. Second, the accuracy of concern identification is evaluated by a controlled experiment with students. Third, we perform seven empirical studies to evaluate the precision of the heuristic detection technique and its supporting tool. Finally, we select three of these studies to understand the empirical correlation of crosscutting patterns and design stability. These evaluation steps are summarised as follows.

- The generality of the conceptual measurement framework is evaluated by using the framework in the instantiation and comparison of several existing metrics. The metrics were selected from different research groups [27, 41, 46, 102, 151] to ensure that they cover a large range of distinct definitions. After each metric is formally described, we also evaluate other elements of the proposed formalism, such as the meta-model and terminology, by formally defining the crosscutting patterns and their detection means.

- The analysis of crosscutting patterns requires the identification of crosscutting concerns in design artefacts. We are aware of limitations in the identification of concerns by developers. To quantify the ability of developers to accurately characterise and identify concerns, we perform three replications of a specific controlled experiment. The hypothesis tested in this experiment is stated as follows: "*the identification of system concerns in design artefacts does not depend on individual differences between developers*".

- We also perform a systematic evaluation of the accuracy of the heuristic detection technique to identify crosscutting concerns. Seven systems from different application domains are used in this evaluation step. We analyse both object-oriented and aspect-oriented designs of such systems, which encompass heterogeneous forms of crosscutting concerns. The accuracy of the heuristic technique is evaluated by comparing its heuristically obtained results with previously available information about the analysed concerns.

- In addition to analysing the accuracy of the heuristic detection technique, we perform an exploratory study on design stability. This evaluation step investigates whether the crosscutting patterns detected by the heuristic technique are good indicators of design stability. In this evaluation, we use detailed design artefacts of two evolving applications comprising 18 successive releases – 8 for the first application and 10 for the second one. In particular, we observe the correlations between certain types of crosscutting patterns and module changes. A similar evaluation is also performed using architecture models instead of detailed design artefacts.

1.5 Outline of the Thesis

In the rest of this document we elaborate on the issues outlined in the introduction. In particular, we present the details of the heuristic assessment technique and its composing elements, such as metrics, detection strategies, and the supporting tool. Based on these elements, we document a set of crosscutting patterns and demonstrate how they can help a developer to identify potential sources of design instabilities. This document is structured as follows.

Chapter 2 provides an overview of the background material necessary to understand the dissertation and evaluate its contributions. Firstly, it presents a definition of design stability and its current assessment strategies (Section 2.1). Chapter 2 also presents the current state-of-the-art in measurement of concern properties (Section 2.2) and metrics-based heuristic analysis (Section 2.3). Moreover, some attempts on the classification of crosscutting concerns are revisited (Section 2.4). Finally, it provides a critical review of the open issues of the state-of-art that motivate this research work (Section 2.5).

In Chapter 3, the foundations of the proposed solution are defined. We present in this chapter the basic terminology and formalism for analysing concern properties (Section 3.1). This formalism includes a meta-model that supports a general and language-independent definition of concerns in software systems. This chapter also proposes a conceptual measurement framework that uses the defined formalism with the purpose of instantiating and comparing concern metrics (Section 3.2). The use of the framework is illustrated by the instantiation of several concern metrics (Sections 3.3

and 3.4) and evaluated by the metrics' application and comparison in the context of a software system (Section 3.5).

Chapter 4 presents and classifies thirteen recurring patterns of crosscutting concerns – i.e., crosscutting patterns – observed in a sample set of software systems. The crosscutting patterns are classified in four groups (Sections 4.2 to 4.5) according to their common underlying characteristics. Each pattern is formalised using our formalism and illustrated by means of an abstract representation and a concrete instance extracted from one of our case studies. This chapter also discusses some of the possible correlations among the crosscutting patterns (Section 4.6).

The heuristic technique to detect crosscutting patterns is presented in Chapter 5. A preliminary step of this technique allows the identification of crosscutting concerns (Section 5.1) which are later classified according to their crosscutting structure. Different strategies and algorithms are used in the identification of crosscutting patterns. These detection strategies and algorithms are individually defined for each group of patterns (Sections 5.2 to 5.5). This chapter also presents our tool, ConcernMorph, which implements the detection strategies and algorithms (Section 5.6).

Chapter 6 presents a preliminary evaluation of our heuristic assessment technique. The first step of this evaluation analyses the accuracy of concern identification based on a controlled experiment with students (Section 6.2). The results of this experiment suggest that a developer can identify concerns in implementation artefacts with a good degree of precision. Based on these results, this chapter also evaluates the usefulness of the heuristic technique to address shortcomings of traditional assessment mechanisms (Section 6.3.2). Finally, it analyses the accuracy of the heuristic technique to identify crosscutting concerns (Section 6.3.3).

Chapters 7 and 8 focus on the evaluation of the proposed assessment technique with respect to architecture and design stability. Three software systems are used in these two chapters. Chapter 7 relies on the heuristic classification of crosscutting patterns in architecture models (Section 7.3) and Chapter 8 performs similar classification in detailed design artefacts. These classifications allow us to measure the number of crosscutting patterns in modules of the target systems (Sections 7.4 and 8.3). These chapters also discuss the correlation between the number of crosscutting patterns and design stability in architecture and later design models (Sections 7.5 and 8.4).

Finally, Chapter 9 summarises the key contributions of this research work (Section 9.1), the lessons learned (Section 9.2), and points out future directions to be followed (Section 9.3).

2. Design Stability and Concern-Oriented Heuristic Analysis

In the first chapter, this research work was motivated and the key areas of interest were defined. These areas include the interplay of measurement and heuristic analysis to support the assessment of design stability. Moreover, we are particularly interested in assessing the correlation of concern properties and design stability. Therefore, we survey existing assessment techniques for this aim, such as concern metrics. Differently from traditional metrics which quantify properties of a particular module, concern metrics quantify properties of concerns with respect to the underlying modular structure [46, 57]. For instance, while traditional metrics can be used to quantify coupling between modules, concern metrics can be used to quantify scattering of a concern over the system modules.

Although undoubtedly useful, as the system grows software metrics are usually too fine-grained and generate too much data which is hard to interpret. Therefore, heuristic analysis [95, 111] has been proposed to support developers with higher level reasoning about the data. This analysis is able to indicate potential design flaws based on the input data. Our main focus is on heuristic analysis to assess design stabilities, and concern metrics play their roles in achieving this aim. That is, the concern metrics are the elementary sources of information feeding the heuristic analysis which, in turn, offers complementary higher-level reasoning of the measurements.

This chapter reviews the literature related to our objectives. Firstly, Section 2.1 discusses the assessment of design stability. It also presents the definition of stability commonly adopted in the literature. Section 2.2 presents an extensive survey of existing concern metrics. Although these metrics come from several research groups, most of them have been used in stability-related assessment activities. Then, Section 2.3 discusses heuristic analysis which can be built on top of metrics. In other words, heuristic analysis is based on metrics and offers heuristic support to capture deviations from good design principles [111]. Section 2.4 reviews some initial attempts at classifying crosscutting concerns. Classification of concerns is important because, as recent studies [44, 73] have shown, crosscutting concerns are not always harmful to design stability. Our heuristic analysis presented later targets at identifying and classifying harmful crosscutting concerns. Section 2.5 presents a critical review of existing concern metrics and techniques for heuristic analysis. For instance, although identification and analysis of concerns are important activities to understand a software design, to the best of our knowledge, there is no study that relies on concern metrics to perform heuristic analysis of design stability. Section 2.6 summarises the discussion of this chapter.

2.1 Assessing Design Stability

Stability is one of the most desirable properties of any software design because it is related to other important quality attributes, such as maintainability and changeability [49, 50, 147]. One widely adopted definition of design stability was first presented by Yau and Collofello [147]. They defined design stability as "*the quality attribute indicating the resistance to the potential ripple effects which a program developed from the design would have when it is modified*". This definition is based on the general knowledge that, as changes are made to a design module other parts of the design may be affected due to the propagation of *ripple effects*. As an example of ripple effect, consider an interface which defines a set of method signatures. These methods are implemented by concrete classes that inherit from this interface. Modifications in this interface, such as adding, deleting, or modifying a method signature, require additional changes to be made to all classes that implement it. Therefore, in this case we say that changes in a particular module have ripple effects in other classes of the design.

In this document, we rely on the above definition of design stability. In particular, a key decision taken in our research was to focus on the assessment of design stability based on module changes which are a direct consequence of ripple effects. Moreover, we investigate design anomalies related to poor modularisation of concerns. This decision was motivated by the fact that previous work [73, 84, 142] has shown that module changes and concern modularisation are strongly correlated. In other words, design instabilities are caused by an iterative bi-directional cycle: concern modularity anomalies require changes to modules which, in turn, lead to new stability-related anomalies. In this research, these facets of design stability are investigated and correlated with the poor modularisation of key design concerns manifested by crosscutting concerns [88]. Crosscutting concerns can hinder design stability because they do not adhere to a system's underlying modular structure. Developers, therefore, may wish to refactor the design to improve modularisation or to implement crosscutting concerns as aspects [88].

Minimising the ripple effects of a program change is a difficult, time consuming, and error-prone activity, especially for badly designed programs. Therefore, some authors [49, 147] have proposed the use of metrics to quantify design stability. Yau and Collofello [147] defined metrics for design stability which indicate the potential ripple effects due to program changes at design level. According to these authors, these metrics can be applied at any point in the design phase of the software life cycle. Elish and Rine [49] investigated the object-oriented metrics proposed by Chidamber and Kemerer [29] as candidate indicators of design stability. In particular, their main goal was to verify whether there are correlations between these traditional metrics and the stability of classes. All these metrics suffer from the limitation of not explicitly considering design concerns in their measurement processes. Despite this limitation, this kind of metrics captures important properties of design stability. Therefore, we advocate that they should not be ruled out in the assessment process, but should be complemented by the recently proposed concern metrics (Section 2.2).

There are initial attempts in the literature to associate concern properties, such as scattering and tangling, with the stability of a system design [44, 73]. For instance, if a concern crosscuts a large number of modules in the design, the effort required to understand and change this concern is expected to be high. Following this direction, this research work also considers the modularity of concerns to assess design stability. Therefore, to complement these initial efforts that rely on the concern abstraction, we propose an approach to predict change-prone modules by using heuristic analysis (Section 2.3). With this analysis, our aim is to support developers by providing them with heuristic means for estimating the stability of a program design. In particular, our research focuses on the correlation of design stability and specific patterns of crosscutting concerns (Section 2.4). In the following section, we briefly describe some concern metrics which have been used in stability-related studies.

2.2 A Survey of Concern Metrics

This section presents a survey of existing concern metrics which have consistently been used in stability-related investigations [27, 41, 46, 102, 133, 134, 151]. Concern metrics have a common underlying characteristic that distinguishes them from traditional modularity metrics [29]: they capture information about concerns realised by one or more modules. Moreover, their definitions allow the measurement of both object-oriented and aspect-oriented designs. Each concern metric of this section is presented in terms of a brief definition and an example. As far as the example is concerned, we rely on a software product line for media manipulation in mobile device applications, called MobileMedia [54, 56]. This product line is further explored in our empirical evaluation (Chapter 7).

An object-oriented design slice of MobileMedia is presented in Figure 2-1. The purpose is to use a common running example to illustrate the application of concern metrics. This figure shows a partial class diagram realising the Chain of Responsibility design pattern [64] which is the concern of interest. The concern metrics are computed based on projecting this concern onto structural elements. For instance, in order to quantify the measurement attributes of the Chain of Responsibility design pattern, elements realising it are shadowed in Figure 2-1.

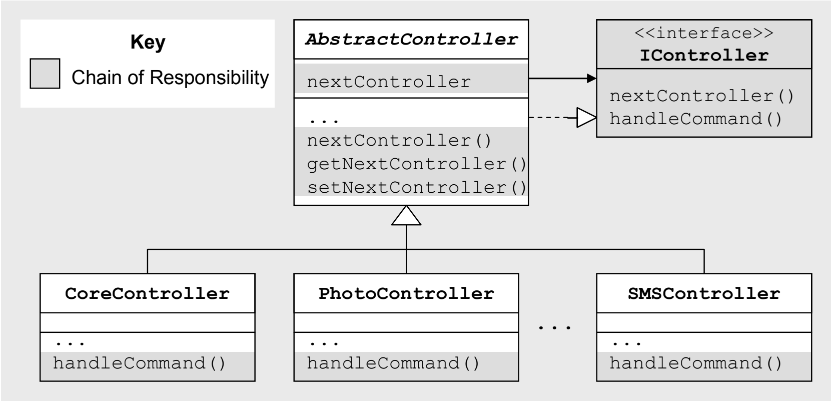


Figure 2-1: Concern projection of the Chain of Responsibility design pattern

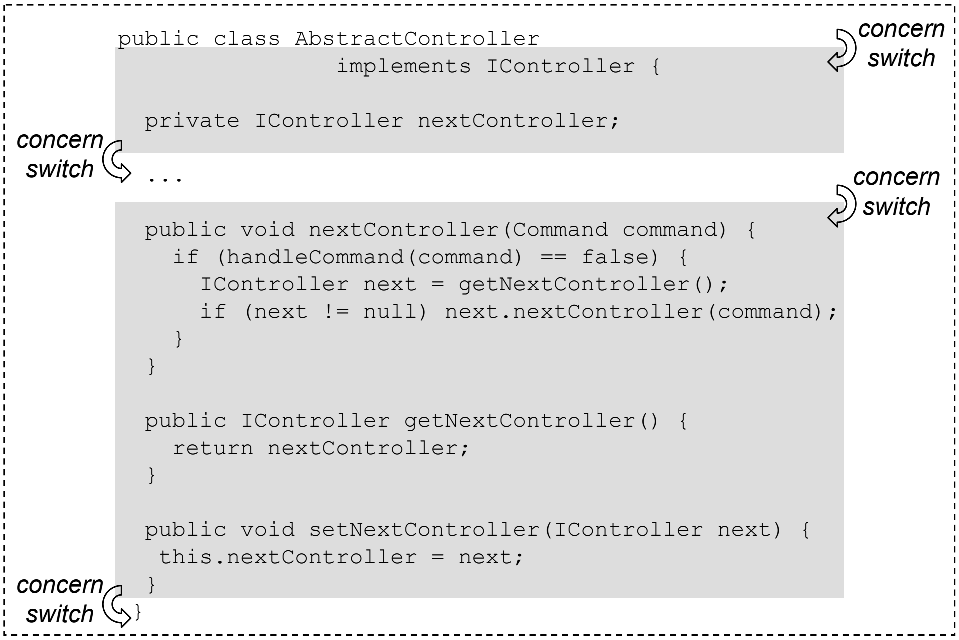
2.2.1 Metrics by Sant'Anna et al. [133, 134]

Sant'Anna and his colleagues [134] defined three concern metrics which quantify the diffusion of a concern over components, operations, and lines of code[2]. Concern Diffusion over Components (CDC) and Concern Diffusion over Operations (CDO) [134] quantify the degree of concern scattering at different levels of granularity. CDC

[2] When the metrics are defined, authors use different terms with similar meanings. For instance, components and operations can be called modules and methods, respectively. In this chapter we keep the same terms used by the original authors.

counts the number of classes and interfaces related to a concern whereas CDO counts the number of methods and constructors. Figure 2-1 shows that behaviour related to the Chain of Responsibility design pattern is spread over five components (CDC) and eight operations (CDO) in that partial design. In another work [133], Sant'Anna *et al.* tailored CDC and CDO for measuring concern scattering in the system architectural description. In architecture models, CDC counts the number of architectural components instead of classes and CDO counts the number of operations in an interface.

In addition to CDC and CDO, these authors defined Concern Diffusion over Lines of Code (CDLOC) which aims at computing the degree of concern tangling. This metric counts the number of "concern switches" for each concern through the lines of code [134]. Concern switches are pairs of lines of code where the first line is related to the implementation of a concern and the following line is part of a different concern. To illustrate this concern metric, Figure 2-2 shows the source code of the `AbstractController` class. This figure indicates that there are four concern switches between Chain of Responsibility and other concerns in the `AbstractController` class. Hence, in this class the value of the CDLOC metric for the Chain of Responsibility concern is four.



```
public class AbstractController
                implements IController {

  private IController nextController;

  ...

  public void nextController(Command command) {
    if (handleCommand(command) == false) {
      IController next = getNextController();
      if (next != null) next.nextController(command);
    }
  }

  public IController getNextController() {
    return nextController;
  }

  public void setNextController(IController next) {
   this.nextController = next;
  }
}
```

Figure 2-2: Projection of Chain of Responsibility in the *AbstractController* class

2.2.2 Metrics by Ducasse, Girba, and Kuhn [41]

Ducasse, Girba, and Kuhn [41] defined a generic technique, called Distribution Map, to analyse *properties* of the system. Using our terminology, we understand “properties” as “concerns”. Based on this technique, they described four concern metrics. In their approach, boxes can represent the program structure (Figure 2-3). For instance, large rectangles, called *partitions*, can be used to represent classes whereas small squares can correspond to internal members of a class, i.e., methods and attributes. Besides, small squares are filled with the colour that represents their corresponding property. Figure 2-3 presents our running example (Figure 2-1) using the Distribution Map notation. The grey squares represent methods and attributes that implement the Chain of Responsibility design pattern.

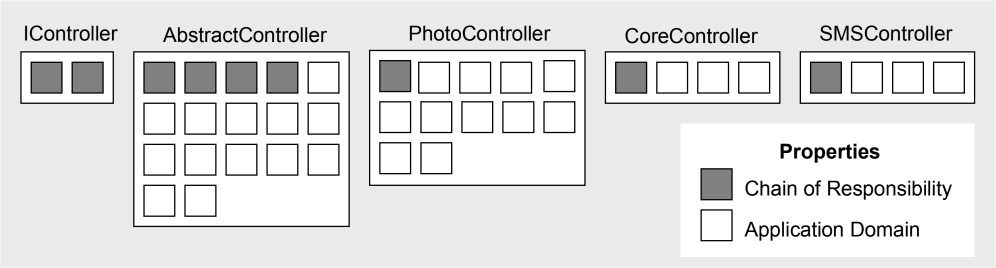


Figure 2-3: Distribution Map showing the Chain of Responsibility property

Based on this representation of the system, four concern metrics were proposed by Ducasse, Girba, and Kuhn [41], namely Size, Touch, Spread, and Focus. The Size metric counts the number of small squares associated with a property. The Touch metric counts the relative size given in terms of the percentage of small squares realising a property. For instance, Figure 2-3 shows that the Size value of Chain of Responsibility is nine and the Touch value is 0.23 (9/39). Spread counts the number of partitions that contains shadowed squares. Note that, in our running example (Figures 2-1 and 2-3), Spread gives the same result (five) as the CDC metric proposed by Sant’Anna [134]. Finally, the Focus metric quantifies the closeness between a particular partition and the property. In other words, the larger the Focus value is, the more parts touched by the property are touched entirely by it. This means that well-encapsulated concerns have a high Focus value whereas crosscutting concerns have a low value. The Focus value of the Chain of Responsibility design pattern is 0.39 in Figure 2-3.

2.2.3 Concentration, Dedication, and Disparity

Wong, Gokhale, and Horgan [151] introduced three concern metrics, namely Disparity, Concentration, and Dedication. Disparity measures how many *blocks* related to a *feature* are localised in a particular component. A feature is the functionality exercised by a given input and a block is a sequence of consecutive statements, so that if one element is executed, all are [151]. Concentration and Dedication which are also defined in terms of blocks quantify how much a feature is concentrated in a component and how much a component is dedicated to a feature [151]. Disparity takes into account the measurements of both Concentration and Dedication. That is, the more blocks either in a component or in a feature, but not in both, the larger the disparity between them.

To illustrate the definitions of Concentration and Dedication, let's consider a component *c* and a feature *f*. Concentration *CONC(f, c)* measures how many of the blocks related to the feature *f* are contained within a specific component *c*. It is defined as follows:

$$\text{CONC(f, c)} = \frac{|\ \text{blocks in component c related to feature f}\ |}{|\ \text{blocks related to feature } f\ |}$$

Similarly, Dedication *DEDI(f, c)* measures how many blocks contained within a component *c* are related to a feature *f*, and is defined as follows:

$$\text{DEDI(c, f)} = \frac{|\ \text{blocks in component c related to feature f}\ |}{|\ \text{blocks in component c}\ |}$$

Inspired by the previously defined concern metrics, Eaddy, Aho, and Murphy [46] presented two variants of Concentration and Dedication. Their concern metrics capture different facets of concern concentration and component dedication to a concern based on lines of code (LOC). In fact, in their metrics, Eaddy, Aho, and Murphy use the term *concern* instead of *feature* and *lines of code* instead of using the concept of *blocks*. Therefore, Concentration is defined as the quotient of LOC in a component realising a concern by the total LOC realising it in the system. Similarly, they also defined the Dedication metric as the quotient of LOC in a component realising a concern by the LOC of the component [46]. In another work, Eaddy and his colleagues [44] defined additional concern metrics, such as Lines of Concern Code (LOCC). LOCC counts the total number of lines of code that contribute to the implementation of a concern.

2.2.4 Basic Concern Metrics

Lopez-Herrejon and Apel [102] defined two of what they called basic concern metrics: Number of Features (NOF) and Feature Crosscutting Degree (FCD). The NOF metric counts the number of features in a system or subsystem. The FCD metric counts the number of classes that are crosscut by a feature. Additionally, Ceccato and Tonella [27] presented a concern metric, called Crosscutting Degree of an Aspect (CDA), which counts the number of *modules* affected by a given *aspect*. FCD [102] and CDA [27] have the same value (five) for the Chain of Responsibility concern in the design of Figure 2-1. Silva *et al*. [136] used a concern metric called Number of Concern Attributes (NOCA) in their refactoring approach. NOCA counts the number of attributes realising a concern in the software system (its value is one in our running example). In the next section we discuss ways of composing metrics to provide developers with higher level interpretation of the results. In particular, we revisit a technique, called detection strategy, proposed by Marinescu [111]. We also discuss the current limitations of concern metrics and detection strategies in Section 2.5.

2.3 Metrics-based Heuristic Analysis

In the previous section, we observed that many concern metrics have been developed to assess different quality attributes of software systems. In particular, some studies [44, 52, 73, 147] have shown that these concern metrics can be used as effective indicators of design stability. That is, they can help to identify potential sources of instabilities in the program design. Despite being a powerful means to evaluate and control the design stability, software metrics also present some problems. For instance, if used in isolation, metrics are often too fine-grained to identify the investigated modularity flaw [95, 111]. The main reason for this is that the interpretation of individual measurements is a hard task. In most cases, individual measurements do not provide relevant clues regarding the cause of a problem. In other words, a metric value may indicate an anomaly in the code but it does not provide the engineer with enough information about the ultimate cause of this anomaly. Therefore, more effort is required in the development of automated means for interpreting the measurements.

In order to cope with this limitation of metrics, some researchers [95, 111, 131] proposed techniques for formulating metrics-based rules that capture deviations from good design principles. Marinescu [111], for instance, proposed a technique called *detection strategies*. A detection strategy is a logical condition composed of metrics which detects design fragments with specific problems. Sahraoui, Godin, and Miceli [131] also used a similar technique and called it *heuristic rules*. Both techniques aim at helping developers to detect design problems and to localise the exact place of these problems in the system design. Using these approaches developers do not need to infer the design problem from a large set of abnormal metric values. Instead, they can directly localise fragments of the design, such as classes or methods, affected by a particular design flaw.

Focusing on Marinescu's detection strategies, the use of metrics is based on *filtering* and *composition* mechanisms [95, 111]. The filtering mechanism reduces the initial data set and, so, only values that present a special characteristic are retained. The purpose for doing that is to detect design elements that have special properties captured by the composing metrics. Marinescu grouped filters into two main categories [111]: absolute filters and relative filters. Absolute filters can be parameterised with a numerical value representing the threshold, e.g., *HigherThan(20)* and *LowerThan(6)* [111]. Relative filters delimit the filtered data set by a parameter that specifies the number of entities to be retrieved, rather than specifying the maximum (or minimum) value allowed in the result set. Thus, the values in the result set are relative to the original set of data, e.g., *TopValues(10)* and *BottomValues(15)*. The relative filters also allow percentage instead of absolute numbers. For instance, *TopValues(25%)* retrieves 25% of the original data set with the highest values.

In addition to the filtering mechanism that limits the interpretation of individual metric values, the composition mechanism supports the association of values from multiple result sets. The composition mechanism is based on set operators that glue together different metrics in an articulated rule [95, 111]. The most common operators used in the composition mechanisms are the *and* and *or* logical operators.

To illustrate this category of heuristic assessment techniques, we present a detection strategy proposed by Marinescu [112] aiming at detecting a specific modularity flaw, named God Class [127]. The God Class problem is described in Riel's book as "an object that knows too much or does too much". God Class represents a class that has grown beyond all logic to become the class that does almost everything [127]. Moreover, it controls too many other objects in the system. The detection strategy for God Class is defined by Marinescu as follows:

```
God Class := (WMC, TopValues(25%)) and (ATFD, HigherThan(1)) and
             (TCC, BottomValues(25%))
```

This detection strategy is based on three traditional metrics for size, coupling, and cohesion, namely Weighted Method per Class (WMC), Access to Foreign Data (ATFD), and Tight Class Cohesion (TCC), respectively. The Weighted Method per Class (WMC) metric measures the sum of the complexity of all methods in a class [29]. The authors rely on McCabe's cyclomatic complexity as the complexity measure [51]. Access to Foreign Data (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods [95]. Tight Class Cohesion (TCC) measures the relative number of directly connected methods [112].

The God Class detection strategy (presented above) says that a class with this problem is among the 25% with highest WMC and has ATFD higher than 1. Moreover, it is among the 25% with lowest TCC. If these three conditions are satisfied, this class represents a potential instance of God Class. To the best of our knowledge, all current detection strategies are based on traditional module-driven metrics. That is, they are restricted to measured properties of modularity units explicitly defined in design languages, such as classes and methods. We further discuss this limitation of detection strategies (and other heuristic assessment techniques) in Section 2.5.2.

2.4 Patterns of Crosscutting Concerns

In addition to the four concern metrics presented in Section 2.2.2, Ducasse, Girba, and Kuhn [41] used the Distribution Map technique to visualise and analyse the concerns of a system. Based on their concern metrics and visualisation technique, they identified four patterns of concerns: Well-Encapsulated, Crosscutting, Octopus, and

Black Sheep. As we said before, Ducasse, Girba, and Kuhn [41] call a concern a *property* and a module a *reference partition*. If a concern corresponds to the reference partitions, they call it Well-Encapsulated. A Well-Encapsulated concern might be spread over one or multiple modules, but almost all elements within those modules are used to realise this concern. Differently from Well-Encapsulated concerns, a Crosscutting concern spreads over several parts and only slightly touches some of them.

According to the definitions given by Ducasse, Girba, and Kuhn [41], Octopus and Black Sheep are in fact specialised categories of a crosscutting concern. Octopus is defined as a concern which is well encapsulated in a few parts, but also spreads across other parts, as does a crosscutting concern [41]. Black Sheep, on the other hand, is a concern that crosscuts the system, but touches very few elements in different modules. To illustrate these two patterns of crosscutting concerns (or *crosscutting patterns*, for short), Figure 2-4 shows a partial representation of the MobileMedia software product line [54, 56] using the Distribution Map notation. Two crosscutting concerns are indicated in this figure: Label and Sorting. The Label concern is an instance of Octopus because it is well encapsulated in `NewLabelScreen`, but it also spreads over several other modules. Moreover, Sorting is an instance of the Black Sheep pattern. That is, it is realised by very few elements in different components. Relying on the Black Sheep metaphor, the system represents the herd and small boxes painted black represent black sheep.

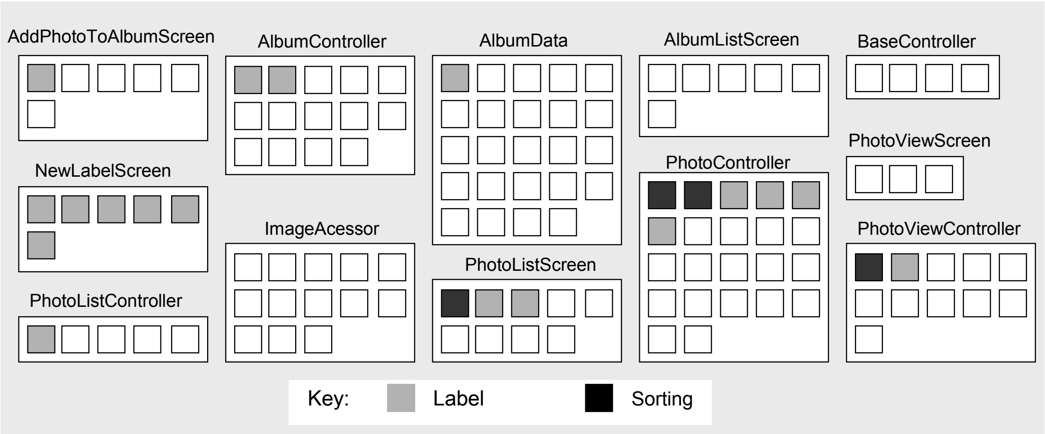


Figure 2-4: Distribution Map showing the Label and Sorting properties

Marin, Moonen, and Deursen [108, 109] have also proposed a classification of crosscutting concerns, called *concern sorts*. According to these authors, concern sorts have the goal of documenting some properties of crosscutting concerns which help their recognition and refactoring towards an aspect-oriented solution. An important property of a concern sort is the aspect language mechanism that supports the modularisation of the concern concrete instance. That is, there is one mechanism of the target aspect-oriented language to address each specific concern sort. In fact, each concern sort is organised according to a template which defines five elements [109]: a name, the intent, an object-oriented idiom, the aspect-oriented mechanism, and examples. The name aims to develop a common language for referring to instances of a concern sort. The intent briefly describes the concern sort. The object-oriented idiom assigned to a concern sort gives its typical Java implementation. Finally, the authors also associate the concern sort with an aspect mechanism that can be used to refactor instances of the concern sort.

Some examples of concern sorts given by Marin, Moonen, and Deursen [108, 109] are Role Superimposition, Policy Enforcement, Exception Propagation, and Declare Throws Clause. Role Superimposition consists of extending the functionality of a class with crosscutting concerns by implementing multiple interfaces. This concern sort is based on the use of *inter-type declarations* [88]. For instance, concerns in this sort can be implemented by adding super-classes or super-interfaces to a class hierarchy by using *declare parents* [87]. Moreover, *inter-type methods* and *inter-type attributes* [87] can also be used to extend the functionality of a class with the concern intent. Policy Enforcement defines a policy which should be applied to the source code. Two AspectJ language constructs, *declare warning* and *declare error* [87], are used to support this concern sort by indicating a policy violation at compile-time. The last two examples of concern sorts, Exception Propagation and Declare Throws Clause, are related to exception handling mechanisms. They rely on AspectJ constructs, such as *exception softening* [87]. It is important to observe that concern sorts [108, 109] are always tied to constructs of a specific aspect-oriented language (i.e., AspectJ [87]).

Hannemann, Murphy, and Kiczales [79] proposed a technique for refactoring crosscutting concerns called Role-based Refactoring. In their approach, crosscutting concerns are described in terms of abstract roles. Then, the instructions for refactoring crosscutting concerns are written in terms of those roles. Their goal is to define abstract refactoring of crosscutting concerns. To enable abstract refactoring steps to be described, Hannemann, Murphy, and Kiczales [79] proposed the definitions of *concrete* and *abstract* crosscutting concerns. According to their definitions, a concrete crosscutting concern is the actual implementation of the concern, while an abstract crosscutting concern description captures the general structure of the concrete crosscutting concerns. For instance, in a typical system, there are multiple cases where the Observer design pattern [64] can be used. These are multiple “Concrete Observer” concerns. All of them have a similar structure which is captured in the “Abstract Observer” concern. The authors only demonstrate the application of their technique in the context of refactoring object-oriented design pattern implementations to aspect-oriented equivalents.

In summary, unlike the patterns of crosscutting concerns defined by Ducasse, Girba, and Kuhn [41], the descriptions of roles [79] and concern sorts [108, 109] present two key problems. First, their definitions are not abstract and generic as required by universal catalogues of patterns. Moreover, roles [79] and concern sorts [108] cannot be applied to either design artefacts or beyond the context of specific programming mechanisms. For example, names for roles [79] are based on a very restricted set of specific low-level concerns (e.g., roles defined by design patterns). In addition, concern sorts [108, 109] are usually tied to constructs of specific AOP languages (e.g., the *declare parents* clause of AspectJ).

2.5 Limitations of the Related Work

This section discusses some limitations of concern metrics (Section 2.5.1) and traditional detection strategies (Section 2.5.2) for evaluating design stability.

2.5.1 Liabilities of Concern Measurement

The extensive survey of concern metrics presented in Section 2.2 supports the claim that analysis of concerns has consistently been used as an important mean for assessing a design. However, this systematic compilation of existing concern metrics also points out fundamental divergences in the manner concerns are currently defined and quantified. One reason for the differences is the distinct objectives pursued by the authors. For example, to investigate maintainability indicators, Wong, Gokhale, and Horgan [151] focus only on the implementation level while Sant’Anna *et al.* [133, 134] examined architecture design, detailed design, and implementation artefacts. Following a distinct approach, Ducasse, Girba, and Kuhn [41] use their own representation of the system. We discuss below three significant differences observed among the concern metrics and their liabilities.

Inconsistent terminology. The non-uniform, distributed manner in which concern metrics are often defined and used results in a lack of standard terminology. Many concern metrics are expressed in an ambiguous manner which limits their use. For instance, the basic modularity unit is called (i) *component* by Sant'Anna *et al*. [133, 134], (ii) *partition* by Ducasse, Girba, and Kuhn [41], and (iii) *module* by Ceccato and Tonella [27]. Similarly, a concern is called *property* [41] and *feature* [151]. This also makes it difficult to understand how different concern metrics relate to each other. In Section 3.1, we propose a terminology and formalism to be used in the analysis and measurement based on the system concerns. The proposed terminology and formalism help, for instance, the comparison of different concern metrics.

Incomplete attributes. In our survey we identified that existing concern metrics target at quantifying four categories of concern properties: scattering, tangling, size, and closeness. Examples of metrics in these categories were presented in Section 2.2. For instance, the Focus metric quantifies closeness, i.e., how close the intention of a design element is in relation to the concern it implements. The metrics Disparity, Concentration, and Dedication, proposed by Wong, Gokhale, and Horgan [151], are also examples of closeness metrics. Furthermore, the concern metrics proposed by Sant'Anna *et al*. [133, 134] target at quantifying scattering and tangling. However, existing metrics (Section 2.2) do not capture the whole spectrum of modularity properties associated with one or more concerns. For example, in spite of the wide recognition that coupling and cohesion play pivotal roles in system maintainability and stability [14, 15, 29, 140], there is no formal metric defined to quantify these concern's modularity properties.

Overlapping measurement goals. There are many different decisions that have to be made when defining a concern metric, such as with respect to the goal of the metric. Unfortunately, for many concern metrics these decisions are not documented. It is, therefore, often unclear what the potential uses of existing metrics are and how different concern metrics could be used in a complementary manner. For instance, the definitions of CDC [134], Spread [41], FCD [102], and CDA [27] are very similar, as indicated by the fact that all these metrics have the value of five for the Chain of Responsibility concern in the running example of Section 2.2 (Figures 2-1 to 2-3). However, without a set of criteria to compare concern metrics, similar measurement goals are not easy to spot and empirical studies relying on the respective metrics cannot be replicated in a reliable fashion [41, 89, 102, 134]. We present in Section 3.2 a conceptual framework which defines a set of criteria to be used in the definition and comparison of concern metrics.

2.5.2 Limitation of Traditional Metrics-Based Heuristic Analysis

Although recent studies [22, 66, 73] have correlated problems of design stability with the inadequate modularisation of concerns, most of the current quantitative heuristic approaches do not explicitly consider concern as a measurement abstraction. This imposes certain shortcomings in the effective employment of metrics to detect design stability impairments. To illustrate the limitations of traditional metric-based heuristic analysis, we investigate the effectiveness of one of Marinescu's detection strategies [111] in the light of a partial design shown in Figure 2-5. This figure presents a partial representation of the object-oriented design of an OpenOrb-compliant middleware system [22]. Elements of the design related to the Observer design pattern [64] are shadowed in this example.

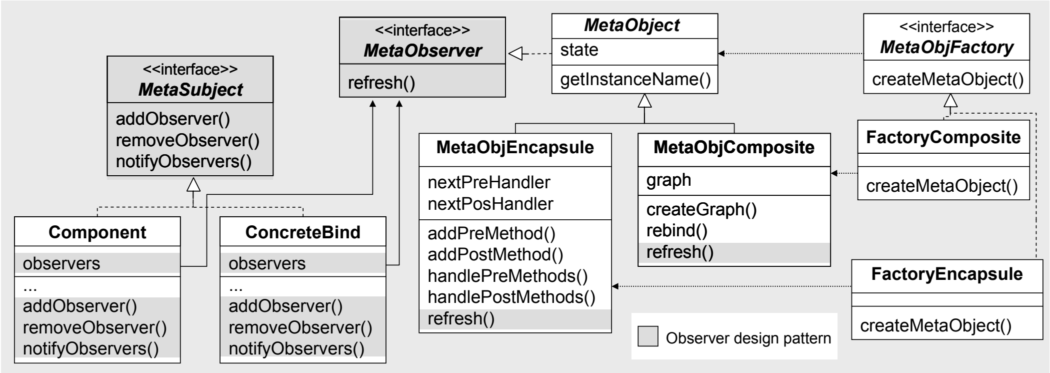


Figure 2-5: The Observer pattern in a partial design model of a middleware system

The analysed detection strategy aims at detecting a specific kind of modularity flaw, namely the Shotgun Surgery bad smell [63]. Bad Smells are proposed by Kent Beck in Fowler's book [63] to diagnose symptoms that may be indicative of something wrong in the design. Shotgun Surgery occurs when a change in a characteristic (or concern) of the system implies many changes in many different places [63]. The reason for choosing Shotgun Surgery in this illustration is because it is believed to be symptomatic of design flaws caused by poor modularisation of concerns [117]. Marinescu's detection strategy for detecting Shotgun Surgery is based on two traditional coupling metrics. This rule is defined as follows [111].

Shotgun Surgery := ((CM, TopValues(20%)) and (CC, HigherThan(5))

CM stands for the Changing Method metric [95], which counts the number of distinct methods that access an attribute or call a method of the given class. CC stands for the Changing Classes metric [95], which counts the number of classes that access an attribute or call a method of the given class. *TopValues* and *HigherThan* are the filtering mechanisms which can be parameterised with values representing the thresholds (Section 2.3). For instance, the Shotgun Surgery detection strategy above says that a class should be in the 20% with highest CM and should have CC higher than 5 to be a suspect case.

Applying CC and CM to our sample design slice, we obtain CC = 0 and CM = 15 for the `MetaSubject` interface (Figure 2-5). Based on these values and computing Marinescu's rule, this interface is not regarded as a suspect of Shotgun Surgery. This occurs because CC is 0, since no class in the system directly accesses `MetaSubject`. Nevertheless, this interface can be clearly considered as Shotgun Surgery because changes to its methods would trigger many other changes in every class implementing it and potentially in classes calling its overridden methods. For instance, a rename of the `addObserver()` method in the `MetaSubject` interface causes updates to the classes `Component` and `ConcreteBind` (Figure 2-5) and several other classes of the system which call `addObserver()`.

This example aims to show how traditional heuristic rules are limited to point out the overall influence of one concern – the Observer design pattern in this case – in other parts of the design. Particularly, the Marinescu's detection strategy was not able to identify that a significant number of classes includes design elements related to the Observer design pattern and, as a consequence, that they could be affected due to a change in this concern. In fact, Marinescu's rule could not highlight the complete impact of the Observer pattern because it considers only metrics based on the module and method abstractions. To the best of our knowledge, all existing detection strategies [95, 111, 112] and similar heuristic techniques [131] are also based on traditional module-driven metrics. In Section 6.3.2, we revisit this example to show that detection strategies composed of concern metrics can identify this design problem.

2.6 Summary

This chapter presented an extensive review of existing quantitative techniques for assessing design stability. It also includes a critical discussion on the limitations of these techniques. Section 2.1 presented the definition of design stability adopted in this document. Based on this definition, Sections 2.2 and 2.3 discuss, respectively, some existing metrics and design heuristic rules, since these are traditionally the fundamental mechanisms for assessing design modularity and stability [29, 51, 95, 111, 127]. In fact, metrics (Section 2.2) are the basic means to assist software developers in characterising and improving the software design. In addition to metrics, there is nowadays an extensive use of metrics-based heuristic analysis (Section 2.3) to capture deviations from good design principles, since metrics are often too fine-grained to comprehensively quantify an investigated quality attribute.

Current quantitative heuristic assessment techniques of design stability usually rely on the traditional module abstractions in order to undertake the measurements. However, recent studies [44, 73, 129] have stressed the value of considering concerns in the design stability assessment process. In particular, there is evidence that some (not all) crosscutting concerns are harmful to design stability [44]. Therefore, we revisited in Section 2.4 the literature related to the classification of crosscutting concerns. For instance, Ducasse, Girba, and Kuhn [41] proposed two crosscutting patterns (Section 2.4), namely Octopus and Black Sheep. Other authors proposed roles [79] and concern sorts [108, 109] to guide the aspect-oriented refactoring of specific types of crosscutting concerns. The latter two approaches could be seen as preliminary steps towards the definition of crosscutting patterns. However, as discussed in Section 2.4, the descriptions of roles and concern sorts (i) are not abstract and generic as required by universal catalogues of patterns and (ii) cannot be applied to either design artefacts or beyond the context of specific programming language mechanisms. Our work complements the existing crosscutting patterns by formalising Octopus and Black Sheep and by defining eleven new patterns (Chapter 4).

Although concern measures have started to be explored in the literature (Section 2.2), the area of concern measurement is still in its infancy and it suffers from not relying on standard terminology and formalisation of concern measurement. Section 2.5 discusses not only the limitations of concern measurements but also the shortcomings of existing metrics-based heuristic analysis. To address some of the limitations of concern metrics, we present in the next chapter a conceptual measurement framework to instantiate and compare concern metrics. Later, in Chapter 5, we also present a heuristic assessment technique based on concern metrics to support the identification of design instability sources. This technique targets at addressing the shortcomings of existing metrics-based heuristic analysis.

3. A Concern-Oriented Conceptual Framework

As discussed in the previous chapter, scattered changes in design modules may be related to poor modularisation of concerns. In fact, there is an increasing awareness that some properties of the system concerns, such as scattering and tangling [73, 88], might be the key factors to the deterioration of design stability. The previous chapter also discussed a number of concern metrics proposed in the literature to support the analysis of concern properties – or *concern analysis*, for short. Concern analysis leads to a shift in the assessment process: instead of analysing properties of a particular module, it focuses on properties of one or multiple concerns and how these concern properties affect the underlying modular structure. One of the goals of concern analysis is to verify whether design flaws are introduced by poor modularisation of concerns. Even though the usefulness of concern analysis is paradigm-agnostic [19, 41, 151], it has been consistently used to support the maintainability assessment of object-oriented designs and their comparison with aspect-oriented designs [46, 52, 56, 61, 67, 73, 102].

However, the area of concern analysis is still in its infancy and it lacks the definition of a conceptual framework. For instance, as discussed in Section 2.5, the terminology used in the definitions of existing concern metrics is diverse and ambiguous. The definitions do not make it clear which is the target level of abstraction ranging from requirements [34] to architecture-level [132] and implementation-specific metrics [46]. It is also unclear which target modularity property, such as high scattering or high coupling, each metric quantifies. They rely on the jargon of specific research groups which hampers: (i) the process of instantiating, comparing, and theoretically validating concern metrics, (ii) independent interpretation of the measurement results, and (iii) replications of empirical studies based on concern metrics.

To address these problems of concern metrics, this chapter presents a conceptual framework for concern analysis. The proposed framework includes a core terminology (Section 3.1) and a set of criteria (Section 3.2) that allow the meaningful definition of well-founded concern metrics. In order to verify the framework applicability and generality, several existing concern metrics are instantiated based on the framework's criteria (Section 3.3). New concern metrics are also proposed and instantiated to match our particular purposes (Section 3.4). After each metric is formally described, we apply and analyse them in the context of two functionally-equivalent versions – one object-oriented and one aspect-oriented – of a software system (Section 3.5). Finally, we compare the selected concern metrics according to (i) their formal definitions and (ii) their results in our empirical assessment. The purpose of formalising and evaluating concern metrics is to verify how and which of them can better support the concern-oriented assessment technique to be presented in Chapter 5.

3.1 Terminology and Basic Formalism

This section presents a terminology to be used in the concern analysis proposed in this document. This terminology allows, for instance, existing concern metrics to be expressed in a consistent and meaningful manner. This chapter focuses on concern measurement but Chapter 4 relies on the same formalism introduced in this section for higher level concern analysis. We seek to define terminology and formalism that are, as much as possible, extensible as well as language- and paradigm-independent [19] which contemplate most current object-oriented and aspect-oriented languages.

3.1.1 Concern-Oriented Meta-Model

Figure 3-1 presents a generic meta-model for concern analysis in the program solution space. This meta-model, described using the UML notation [121], is based on previously defined meta-models for measurement [5, 14, 15]. We also survey meta-models for aspect-oriented programming languages [28, 77, 101]. However, for simplicity reasons, the proposed meta-model only defines elements that are relevant to the scope of this thesis. We keep our meta-model simple in order to achieve greater generalisation.

An instance of the meta-model presented in Figure 3-1 is called a *system*. An *element* is either a *component*, an *interface*, or a *member*. The meta-model defines not only the possible relations between concerns and the system's structure, but also the key abstractions for module specifications. In other words, it describes the mapping of structural elements to concerns at the level of abstraction being assessed. For example, when applied to design specifications the meta-model describes design artefacts. On the other hand, it would describe the source code when used at the implementation level. The meta-model can also be instantiated to represent elements of aspect-oriented languages. In this case, the concern analysis is supposed to consider the design structure before any automated processing, like compilation or weaving.

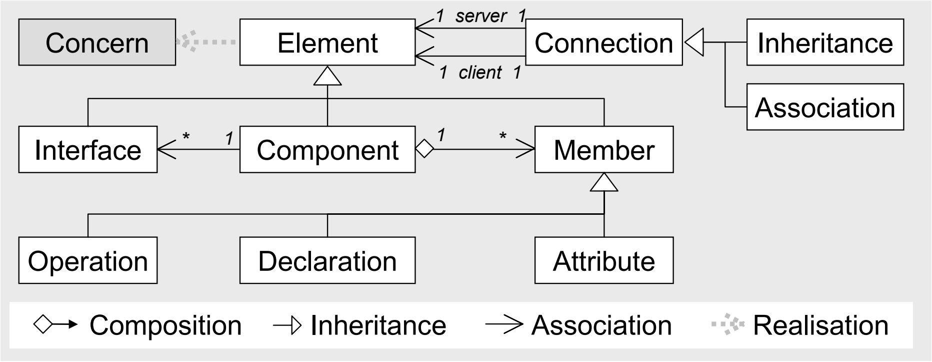


Figure 3-1: Meta-Model of Concern Realisation

Due to its generality, sometimes it is required to decorate the meta-model with additional information. For example, implementation-level metrics commonly use the number of lines of code as abstraction. However, we do not represent lines of code in our generic meta-model because (i) they are too specific to the system's implementation and (ii) they suffer from variations in, for instance, the programming style and the used editor. This issue can be addressed though by decorating the elements of our meta-model with an implementation-specific LOC attribute. For instance, the set of all lines of code of a system *S* at the implementation level is denoted by *LOC(S)*. Similarly, the sets of lines of code of a component *c* and of a concern *con* are defined by *LOC(c)* and *LOC(con)*, respectively. The rest of this section thoroughly discusses the key properties of the proposed meta-model.

3.1.2 Components and Concerns

As indicated in the meta-model of Figure 3-1, a *concern* can be realised by an arbitrary set of elements. A system *S* consists of a set of components, denoted by *C(S)*. The sets of all attributes, operations, and declarations of the system *S* are denoted by *Att(S)*, *Op(S)*, and *Dec(S)*, respectively. A declaration is usually defined by aspect-oriented languages. For instance, it can be either a 'declare parent' or a 'pointcut declaration' in AspectJ [87] (see Section 3.1.4). As defined by the meta-model, a component *c* has a set of interfaces, *I(c)*; the set of interfaces of a system *S* is defined by *I(S)*. Besides, each component *c* consists of a set of attributes, *Att(c)*, a set of operations, *Op(c)*, and a set of declarations, *Dec(c)*. The set of members of a component *c* is defined by *M(c) = Att(c)* ∪ *Op(c)* ∪ *Dec(c)*.

The reader should notice that, for generality purposes, a component is a unified abstraction to capture both aspectual and non-aspectual modules. The idea is to make the meta-model paradigm- and language-independent [19]. For example, a component represents either a class or an interface in Java programs [38], and a component is a class, an interface, or an aspect in AspectJ programs [87].

On top of this structure, we define *concerns* which are selections of any type of elements (Figure 3-1). A concern is not directly mapped to an element in a design model or program, such as components and operations. However, a concern can be considered an abstraction which is realised by those elements that have the purpose of realising it. An example of concern is a software requirement. In this way, in order to have concern analysis it is necessary to identify the structural elements (e.g., components) realising a concern.

The set of concerns realised by the system *S* is defined as *Con(S)*. A concern *con* can be realised by a set of components, *C(con)*, and the set of concerns addressed by a component *c* is defined as *Con(c)*. The set of switches (Section 2.2.1) related to a specific concern *con* is given by *Switches(con)* and the set of blocks (Section 2.2.3) assigned to a concern *con* is defined by *Blocks(con)*. A concern *con* can also be realised by a set of attributes, *Att(con)*, a set of operations, *Op(con)*, or a set of declarations, *Dec(con)*. The set of members that realise a concern *con* is defined as *M(con) = Op(con)* ∪ *Att(con)* ∪ *Dec(con)*. In fact, a concern can be realised by every element – just a single element or a collection of different elements – of the meta-model presented in Figure 3-1.

3.1.3 Connections and Component Sets

In addition to structural elements, the proposed formalism also considers relationships, such as connections and inheritance relationships [14, 15], among the meta-model's elements. Basically, *connection* is defined as a dependency relationship between two elements, where one element, called *Server*, provides a service to another element, the *Client* [14]. For instance, a connection of a component *c* can be caused by elements of *c* calling an operation or accessing an attribute of another component. The set of connections of a component *c* is defined as *CC(c)* and the set of connections of a member *m* is defined as *CC(m)*. A connection is assigned to a concern when both client and server elements also are. The set of connections assigned to a concern *con* is defined by *CC(con)*.

For convenience, we define that the set of all components of a system, *C(S)*, can be partitioned into 3 subsets: Application, *AC(S)*, Framework, *FC(S)*, and Library, *LC(S)*. Furthermore, components may participate in inheritance relationships. For a given component *c*, the following sets are defined: (i) *Ancestors(c)* - all recursively defined parents; (ii) *Parents(c)* - the directly defined parents; (iii) *Children(c)* - the directly derived children, and (iv) *Descendants(c)* - the recursively derived children. Because of inheritance relationships between components, the following member sets are defined for a component *c*:

$M_{NEW}(c)$ - Members newly implemented in the component (not inherited);

$M_{VIR}(c)$ - Members declared as virtual/abstract;

$M_{OVR}(c)$ - Members inherited and overridden;

$M_{INH}(c)$ - Members inherited and not-overridden.

As previously observed, the structure of the system is considered before any kind of automated processing. This reflects how the terminology described in this section should be interpreted. For instance, in the case of aspect-oriented languages [87, 114], member sets of a component represent the structure before aspect weaving. In other words, members introduced by intertype declarations (e.g., in AspectJ [87]) are not accounted for their target components (i.e., classes), but for their declaring components (i.e., aspects). Similarly, inheritance relationships are analysed before any inheritance manipulation is applied, such as AspectJ's declare parents [87].

3.1.4 Meta-Model Instantiation

The aforementioned structures are abstract enough to be instantiated for different modelling and programming languages. This section provides a brief illustration on how our generic meta-model (Figure 3-1) can be instantiated to languages targeting different purposes and levels of abstraction. We use an architecture modelling language for the component-and-connector view [6], and two programming languages, namely Java and AspectJ [87]. We choose Java as a representative of object-oriented languages, AspectJ as a representative of aspect-oriented programming languages, and Component & Connector models as an example of a modelling language used in early design stages. We choose these languages for illustration because they are widely used in practice for different purposes. Table 3-1 defines how some elements of the meta-model can be mapped to these three languages. A blank cell means that the abstraction is not implemented by any element of the language. For example, declarations are usually valid only for aspect-oriented languages, such as AspectJ (Table 3-1).

Table 3-1: Meta-model instantiation.

	Component & Connector [6]	Java	AspectJ [87]
System	Architecture Design	Java System	AspectJ System
Concern	Architecture Concern	Concern	Concern
Component	Architecture Component	Class and Interface	Class, Interface, and Aspect
Interface	Architecture Interface	Method Signatures	Method Signatures
Attribute	-	Class Variable and Field	Class Variable, Field, and Inter-type Attribute
Operation	Operation and Event	Method and Constructor	Method, Constructor, Inter-type Method and Constructor, and Advice
Declaration	-	-	Pointcut and Declare Statement

3.2 On the Criteria to Be Used for Concern Metric Instantiation

Concern metrics are the elementary means to characterise concern properties. This section presents a conceptual framework for instantiation and comparison of concern metrics (or *concern measurement framework* from now on). This framework relies on the terminology and formalism presented in the previous section. It is important to stress that the proposed terminology and formalism are intended to support concern analysis beyond measurements. For instance, they are also used in the formalisation of crosscutting patterns (Chapter 4) and in the proposed heuristic assessment technique (Chapter 5).

In order to define a concern measurement framework, we analysed and, whenever it was feasible, tried to maximise the use of criteria defined in existing measurement frameworks for object-oriented [14-16, 89] and aspect-oriented [5] systems. Moreover, we have identified recurring characteristics of existing concern metrics proposed in the literature (see Section 2.2). From these analyses the following set of criteria has emerged. These criteria should be considered when either comparing concern metrics or developing a new concern metric.

1. Entity of concern measurement
2. Concern-aware property
3. Unit and scale type
4. Properties of values
5. Concern granularity
6. Domain
7. Concern projection

The next subsections provide details on each criterion. For each criterion, we present a definition and some possible options that can be selected. We also discuss how some existing concern metrics (Section 2.2) could be defined with respect to that specific criterion.

3.2.1 Entities of Concern Measurement

One of the goals of concern measurement is to capture properties of concerns, such as size, in software artefacts and express them in a formal way. The entity of measurement determines the elements that are going to be measured. When we choose a certain element type as the entity of measurement, it means that we are interested in properties of this type. For example, if we choose component, it means we are interested in concern-related information of components.

Criterion Instantiation. Usually, concern metrics define concerns as the entity of measurement, but other selections are also possible. For instance, the metrics Concentration and Dedication [151] discussed in Section 2.2 have different entities of measurement. While Concentration has concerns as entities, the Dedication metric is interested in information of components. Although all elements in the meta-model of Figure 3-1 may be selected in this criterion, the most common entities of concern measurement are: (i) System, (ii) Concern, and (iii) Component.

3.2.2 Concern-aware Properties

Properties are the characteristics an entity of measurement (e.g., concern) possesses. For a given property, there is a relationship of interest in the empirical world that we want to capture formally in the mathematical world [89]. For example, if we observe two concerns we can say that one *is more scattered than* the other. A concern metric allows us to captures the "is more scattered than" relationship and maps it to a formal system. This enables us to explore the relationship mathematically. An entity of measurement possesses many properties, while a property can characterise many different entities [89]. These relationships can be confirmed by example. To see that an entity can have many properties, consider a concern as an entity of measurement which can exhibit properties such as scattering and tangling. In addition, a property may apply to one or more different entities. For instance, size can be applied to several different software entities, such as components or concerns.

Criterion Instantiation. In the selection of a concern-aware property, we may choose any characteristic of the entity that we want to measure. In fact, existing concern metrics in the literature cover a vast range of measurement properties. For example, the metrics proposed by Sant'Anna [134] (Section 2.2) target *scattering* (CDC and CDO) and *tangling* (CDLOC). On the other hand, Wong's metrics Concentration, Dedication, and Disparity [151] assess the *closeness* between components and concerns. Possible values of a measurement property include: (i) Scattering, (ii) Tangling, (iii) Closeness, (iv) Coupling, (v) Cohesion, and (vi) Size.

3.2.3 Units and Scale Types

The *concern measurement unit* criterion determines how we measure a property. A property may be measured in one or more units, and the same unit may be used to measure more than one property [89]. For example, concern size might be measured by counting either lines of code or components which realise the concern. Similarly, the number of components may be used to measure the size and scattering properties of a concern.

When we consider measurement units, we also need to understand the different s*cale types* implied by the particular unit. The most common scale types [141] are: nominal, ordinal, interval, and ratio. A unit's scale type determines the admissible transformations we can apply when we use a particular unit [51]. In the *nominal* scale [141], names are assigned to entities as labels. For example, a concern can be labelled as "well-encapsulated" or "crosscutting" in the nominal scale. The patterns of crosscutting concerns, such as Black Sheep and Octopus, we discussed in Section 2.4 are also examples in the nominal scale. Unlike the nominal scale, the *ordinal scale* [141] represents the rank order (1st, 2nd, 3rd, etc.) and numbers are assigned to entities of measurement. This scale allows a comparison between two concerns, for example, to characterise which of them is more crosscutting.

In the *interval scale* [141], the numbers assigned to entities have all the features of ordinal measurements and, in addition, equal differences between measurements represent equivalent intervals. That is, differences between arbitrary pairs of measurements can be meaningfully compared. Therefore, operations such as addition and subtraction are meaningful. Finally, in the *ratio scale* [141] the numbers assigned to entities have all the features of interval measurement and also have meaningful ratios between arbitrary pairs of numbers. Operations such as multiplication and division are therefore meaningful. Interval and ratio are the most commonly used scale types in the concern metrics (and measurement in general [89]).

Criterion Instantiation. The concern metrics discussed in Section 2.2 use interval or ratio scale types, but they have heterogeneous units of measurement. For instance, the metrics CDC, CDO, and CDLOC [134] use the interval scale type and their units are the number of components, operations, and concern switches, respectively. On the other hand, all concern metrics proposed by Wong, Gokhale, and Horgan [151] use the ratio scale and none of them have units of measurement. The possible values for scale types are [89] (i) Nominal, (ii) Ordinal, (iii) Interval, and (iv) Ratio. We may choose any countable elements as measurement units, for example, (i) Concerns, (ii) Components, (iii) Operations, (iv) Attributes, and (v) Concern Switches.

3.2.4 Concern Measurement Values

A measured value cannot be interpreted unless we know to what concern it applies, what property it measures and in what unit. Some concern metrics, such as those proposed by Wong, Gokhale, and Horgan [151], do not specify any unit of measurement. The lack of units in the metrics investigated in Section 2.2 is a result of equations which divide two values with the same unit, e.g., the Touch metric divides members (realising a concern) per members (of the system).

Criterion Instantiation. We expect concern metrics to be defined over a set of permissible values. Nominal and ordinal scale measures are usually discrete (i.e., map to integers), whereas interval and ratio measures can be continuous or discrete. For instance, the CDC [134] is an interval-scale measure defined on the non-negative integers. On the other hand, values of Concentration, Dedication and Disparity [151] are continuous and bounded in the range of 0 and 1, inclusive. A set of permissible values may be Finite or Infinite, Bounded or Unbounded, Discrete or Continuous.

3.2.5 Concern Granularity

The granularity of a concern metric is the level of detail at which information is gathered. This criterion is determined by two factors: (i) *element granularity* – what elements are measured, and (ii) *element distinction* – how the elements are counted. The element granularity factor specifies what is being counted, i.e., which elements aggregate values. For example, when we say "*the number of concerns of a component that...*" the entity is component but what we are counting (granularity) is concern.

The element distinction factor defines whether we ignore duplicated elements or not when re-applying the concern metric to a different goal. For instance, it may be allowed to count the same operation for different concerns in a given metric (e.g., CDO [134]) but not for others (e.g., Touch [41]). The difference between element granularity and measurement unit is clear because all metrics have to define an element which is being counted. However, the measurement unit can either be omitted or be coarser or finer than the granularity (e.g., giving weights to elements). For instance, we could define a metric as the quotient between components realising a concern and the number of components of the system. Although this hypothetical metric does not specify any unit of measurement, its granularity is still component.

Criterion Instantiation. Our survey of concern metrics (Section 2.2) points out very heterogeneous options for element granularity and element distinction. For instance, element granularity ranges from lines of code (e.g., CDLOC) to components (e.g., CDC) and concerns (e.g., NOF). There are also metrics which allow elements to be counted more than once, such as CDC and CDO [134], and metrics that allow each element to be counted only once, such as Size and Touch [41]. Possible values of granularity are, for instance: (i) Concern, (ii) Component, (iii) Operation, (iv) Attribute, and (v) Lines of Code. Element distinction has to be Yes (count only once) or No (count all possible occurrences).

3.2.6 Domain

There are two pertinent factors to be considered for domain: *how to account for inheritance* and *how to restrict types of elements for being measured*. Regarding inheritance, concern metrics have to define how components related via inheritance should behave. For instance, if inherited operations should be excluded or included in a given concern metric. In the domain criterion, metrics have also to define the types of elements that should be accounted for. For example, they might strictly count elements of the application domain (excluding classes of the framework and libraries).

Criterion Instantiation. The possible values for inheritance are Yes (account) or No (ignore). Additionally, if inheritance is taken into consideration, a metric has to specify which set of elements should be included, e.g., Ancestors, Parents, Children, or Descendants. We may restrict elements in the domain based on: Application, Framework, and Libraries. Other categorisations are also conceivable. However, we suggest using a classification scheme where the decision into which category a given element belongs can be made automatically. Based on the original definitions of the analysed concern metrics (see Section 2.2), we cannot decide if the metrics take inherited members into consideration or not and how they restrict the domain. Therefore, we acknowledge that the investigated concern metrics do not consider inheritance and that they are applied to application elements only.

3.2.7 Concern Projection

Concern metrics must specify a measurement protocol to be followed in empirical studies; otherwise, these empirical studies cannot be replicated. Study replication is the basis of scientific validation [89]. A measurement protocol must be unambiguous and must prevent invalid measurement procedures such as double counting. One of the most sensitive parts in concern measurement is the projection of abstract concerns onto elements of the design and implementation. At least two definitions have to do with this projection: (i) *what the concerns are* and (ii) *onto what artefacts the concern is supposed to be projected*. Moreover, concern metrics have to specify whether *they allow overlapping of concerns* or not. For instance, can two different concerns be projected onto the same operation?

Criterion Instantiation. Most of the concern metrics do not clearly state which sorts of concerns they are interested in. When the specification of a concern is not clear in the metric definition, we assume it is applied to all kinds of concerns. Typical concerns in a software project include [129]: (i) Features, (ii) Non-Functional Requirements, (iii) Design Patterns or Idioms, and (iv) Implementation Mechanisms (e.g., caching). Examples of design elements which concerns can be projected onto are (i) Components, (ii) Operations, (iii) Attributes, and (iv) Lines of Code. All three concern metrics proposed by Sant'Anna (i.e., CDC, CDO, and CDLOC) are applied to all kinds of concerns. However, a concern is projected onto different artefacts: CDC requires a projection of concerns onto components, CDO onto operations, and CDLOC onto lines of code. Of course, a projection onto finer-grained elements (e.g., operations) can easily be extended to coarser-grained ones (e.g., classes). In other words, the CDC metric also accepts a projection of concerns onto operations or lines of code since the projection of classes can be easily derived from that.

3.3 Application of the Framework

This section demonstrates how to instantiate and formalise concern metrics using the proposed framework. Moreover, we want to verify the following hypothesis in this section.

> ***Hypothesis 3-1****: Existing sets of concern metrics are incomplete in the sense that they do not cover all relevant concern properties.*

To verify this hypothesis, we follow two steps. First, we illustrate the application of our framework by describing a concern metric proposed by Sant'Anna [134] (Section 3.3.1). Second, we systematically apply the framework to several sets of existing concern metrics (Section 3.3.2). We then evaluate both (i) the framework applicability to define unambiguous and fully operational concern metrics and (ii) its ability to pinpoint weaknesses of these metrics.

3.3.1 Step by Step Metric Formalisation

This section demonstrates the application of our concern measurement framework to instantiate the metric Concern Diffusion over Operations (CDO) [134]. This metric has been applied to measure design stability in a number of empirical studies [67, 73, 56]. We first describe the metric textually using the proposed measurement terminology (in *italic*). Then, we go through the whole process of analysing and selecting each criterion of the measurement framework and, finally, we derive a formal definition.

Description. Concern Diffusion over Operations (CDO) counts the number of *operations* in a *component* whose purpose is to realise a *concern*.

Considering the set of framework criteria (Section 3.2), we select the options that match the CDO definition as follows.

1. **Entity of Concern Measurement**. *Concern* is the entity of measurement for this metric.

2. **Property**. CDO quantifies *scattering* of a given concern over the operations of the system.

3. **Unit and Scale Type**. The measurement unit is *number of operations* and it is measured in an *interval scale*.

4. **Properties of Values**. Permissible values for this metric are not greater than *Op(S)* (*finite*), defined in the interval between 0 and Op(S) (*bounded*), and allow integers only (*discrete*).

5. **Granularity**. The granularity of elements that are being measured is *operation*.

6. **Domain**. It considers *application components* (not components in the framework or libraries) and *does not take inherited operations into account*. That is, it considers operations in $M_{NEW}(c) \cup M_{VIR}(c) \cup M_{OVR}(c)$.
7. **Concern Projection**. Concerns can be *anything* that directly contribute to the functionality of the system, i.e., which is materialised in the design or implementation. It requires projection of concerns onto *operations* (or finer-grained artefacts). *Overlapping* of concerns is allowed.

Using the selected criteria and the set notation introduced in Section 3.1 we derive the following formal definition for CDO.

For a given concern $con \in Con(S)$ and $c \in C(S)$:

$$\text{CDO(con, c)} = \left| \text{Op(c)} \cap \text{Op(con)} \right|$$

This formula states that CDO(con, c) is given by the number of elements in the intersection $Op(c) \cap Op(con)$. This intersection selects operations which realise the concern *con* in the component *c*. The value of CDO(con) for a given concern *con* is the summation of CDO(con, c) for every component $c \in AC(S)$.

3.3.2 Instantiation of Concern Metrics

In addition to CDO illustrated in Section 3.3.1, we have instantiated and formally defined other 16 concern metrics using the proposed measurement framework. We aim to formalise and compare them according to the framework's definition. These metrics were selected for several reasons. First, they were proposed by different authors in various research groups. Hence, they do not rely on a uniform terminology. Second, the selected metrics present a very heterogeneous definition, apply to distinct levels of abstraction ranging from architecture to implementation, and quantify diverse assessment goals. Finally, these concern metrics have been applied in a number of maintenance empirical studies [41, 56, 61, 66, 73, 133] and they seem to be related to external quality attributes, such as design stability.

Tables 3-2, 3-3, and 3-4 show the selected criteria for the sample set of concern metrics. Rows of these tables list the framework criteria and columns present the concern metrics. Table 3-5 presents the formal definitions of these concern metrics according to the proposed framework. These tables highlight, for instance, the need for concern metrics which contemplate other domains (criterion 6) rather than the application one. For example, Briand *et al*. [14] state a hypothesis that "*maintainability is influenced by dependencies on both stable (library) and unstable classes (application)*". To verify this hypothesis, one might use concern metrics to evaluate the influence of library concerns in the design of the system. None of the existing concern metrics explicitly deal with inheritance which indicates the lack of concern metrics covering some alternative options of the framework criteria.

Table 3-2: Instantiation of concern metrics from Sant'Anna, Silva, and Lopez-Herrejon.

	Sant'Anna *et al.* [134]		Silva *et al.*	Lopez-Herrejon and Apel [102]	
	CDC/CDA	CDLOC	NOCA [136]	NOF	FCD
1. Entity	Concern	Concern	Concern	Component	Concern
2. Property	Scattering	Tangling	Size	Tangling	Scattering
3. Unit and Scale	Components Interval	Switches Interval	Attributes Interval	Concerns Interval	Components Interval
4. Values	*[0, C(S)]* Discrete	*[0, ∞]* Even Values	$[0, \sum^{c \in C(s)} Att(c)]$ Discrete	*[0, Con(S)]* Discrete	*[0, C(S)]* Discrete
5. Granularity and Distinction	Component No	Line of Code No	Attribute Yes	Concern Yes	Component No
6. Domain and Inheritance	Application No	Application No	Application No	Application No	Application No
7. Concern, Artefact and Overlapping	All Component Yes	All Line of Code No	All Attribute Yes	Feature Component Yes	Feature Component Yes

Table 3-3: Instantiation of concern metrics from Ducasse and Eaddy.

	Ducasse, Girba, and Kuhn [41]				Eaddy *et al.*
	Size	Touch	Spread	Focus	LOCC [44]
1. Entity	Concern	Concern	Concern	Concern	Concern
2. Property	Size	Size	Scattering	Closeness	Size
3. Unit and Scale	Members Interval	None Ratio	Components Interval	None Ratio	Lines of Code Interval
4. Values	$[0, \sum^{c \in C(s)} M(c)]$ *D*iscrete	*[0, 1]* Continuous	*[0, C(S)]* Discrete	*[0, 1]* Continuous	*[0, LOC(S)]* Discrete
5. Granularity and Distinction	Member Yes	Member Yes	Component No	Member Yes	Line of Code Yes
6. Domain and Inheritance	Application No	Application No	Application No	Application No	Application No
7. Concern, Artefact and Overlapping	All Member No	All Member No	All Component No	All Member No	All Line of Code Yes

Table 3-4: Instantiation of concern metrics: Concentration, Dedication, and Disparity.

	Wong *et al*. [151]			Eaddy *et al*. [46]	
	Disparity	Concentration^{1}	Dedication^{1}	Concentration^{2}	Dedication^{2}
1. Entity	Concern Component	Concern	Component	Concern	Component
2. Property	Closeness	Closeness	Closeness	Closeness	Closeness
3. Unit and Scale	None Ratio	None Ratio	None Ratio	None Ratio	None Ratio
4. Values	*[0, 1]* Continuous	*[0, 1]* Continuous	*[0, 1]* Continuous	*[0, 1]* Continuous	*[0, 1]* Continuous
5. Granularity and Distinction	Member No	Member No	Member No	Line of Code No	Line of Code No
6. Domain, Inheritance	Application No	Application No	Application No	Application No	Application No
7. Concern, Artefact, and Overlapping	Feature Member Yes	Feature Member Yes	Feature Member Yes	All Line of Code No	All Line of Code No

Table 3-5: Formal definitions of concern metrics.

$\mathrm{CDC(con)} = \mathrm{CDA(con)} = \lvert \mathrm{C(con)} \rvert$	$\mathrm{Spread(con)} = \lvert \mathrm{C(con)} \rvert$
$\mathrm{CDLOC(con)} = \lvert \mathrm{Switches(con)} \rvert$	$\mathrm{LOCC(con, c)} = \lvert \mathrm{LOC(con)} \cap \mathrm{LOC(c)} \rvert$
$\mathrm{NOCA(con, c)} = \lvert \mathrm{Att(con)} \cap \mathrm{Att(c)} \rvert$	$\mathrm{Concentration(con, c)}^{1} = \frac{\lvert \mathrm{Blocks(con)} \cap \mathrm{Blocks(c)} \rvert}{\lvert \mathrm{Blocks(con)} \rvert}$
$\mathrm{NOF} = \lvert \mathrm{Con(S)} \rvert$	$\mathrm{Dedication(con, c)}^{1} = \frac{\lvert \mathrm{Blocks(con)} \cap \mathrm{Blocks(c)} \rvert}{\lvert \mathrm{Blocks(c)} \rvert}$
$\mathrm{FCD(con)} = \lvert \mathrm{C(con)} \rvert$	$\mathrm{Disparity(con, c)} = \frac{\lvert \mathrm{Blocks(con)} \cap \mathrm{Blocks(c)} \rvert}{\lvert \mathrm{Blocks(con)}\ \mathrm{U}\ \mathrm{Blocks(c)} \rvert}$
$\mathrm{Size(con)} = \lvert \mathrm{M(con)} \rvert$	$\mathrm{Concentration(con, c)}^{2} = \frac{\lvert \mathrm{LOC(con)} \cap \mathrm{LOC(c)} \rvert}{\lvert \mathrm{LOC(con)} \rvert}$
$\mathrm{Touch(con, c)} = \frac{\lvert \mathrm{M(con)} \cap \mathrm{M(c)} \rvert}{\lvert \mathrm{M(c)} \rvert}$	$\mathrm{Dedication(con, c)}^{2} = \frac{\lvert \mathrm{LOC(con)} \cap \mathrm{LOC(c)} \rvert}{\lvert \mathrm{LOC(c)} \rvert}$
$\mathrm{Focus(con, c)} = \frac{\lvert \mathrm{M(con)} \cap \mathrm{M(c)} \rvert}{\lvert \mathrm{M(c)} \rvert} * \frac{\lvert \mathrm{M(con)} \cap \mathrm{M(c)} \rvert}{\lvert \mathrm{M(con)} \rvert}$	

It can be noted in Tables 3-2, 3-3, and 3-4 that the selected concern metrics present heterogeneous values in some criteria, such as property and measurement unit. For instance, existing concern metrics in the literature cover at least four properties of concerns, namely Scattering, Tangling, Size, and Closeness. However, to the best of our knowledge, none of the existing concern metrics target other equally important concern properties, such as coupling and cohesion. This observation confirms our hypothesis stated in the beginning of this section. That is, existing concern metrics do not cover all relevant concern properties. Since the hypothesis was not satisfied, we propose and instantiate new concern metrics addressing coupling and cohesion in Section 3.4. Section 3.5 provides further comparison of the concern metrics instantiated in this section after applying them to quantify modularity and stability of an evolving system.

3.4 Beyond Existing Concern Metrics

This section verifies whether the measurement framework is helpful to define new concern metrics to different evaluation goals. It instantiates and formalises four new concern metrics: two coupling and two cohesion metrics. The reason for creating these concern metrics is because they are required by our assessment technique (Chapter 5). These metrics provide extra pieces of information about concerns, such as coupling between concerns and component cohesion based on the concern realisation.

For coupling, we define two new metrics called Concern-Sensitive Coupling (CSC) and Intra-Component Concern-Sensitive Coupling (ICSC). Additionally, we define the metrics Lack of Concern-based Cohesion (LCC) and Concern-Sensitive Component Cohesion (CSCC) for cohesion. These concern metrics address two concern properties (coupling and cohesion) in the set of criteria which are not covered by existing concern metrics. We textually describe below these concern metrics and present their formal definitions. Italic text highlights the words compliant to the proposed terminology, such as the elements of the meta-model presented in Section 3.1. Table 3-6 shows the metrics' instantiations according to the framework set of criteria. Rows of this table list the framework criteria while columns depict the proposed concern metrics.

Concern-Sensitive Coupling (CSC) quantifies the number of *connections* assigned to the *concern* and used by a *client component* to request services from *server components*.

$$\mathrm{CSC}(con, c) = \left| \mathrm{CC}(c) \cap \mathrm{CC}(con) \right|$$

Intra-Component Concern-Sensitive Coupling (ICSC) counts the number of *members* of a *component* accessed by other *members*. Caller members realise the concern and callee members do not.

$$\mathrm{ICSC}(con, c) \;=\; \sum^{m \in M(c)} \mathrm{ICSC}(con, m)$$

$$\mathrm{ICSC}(con, m) \;=\; \left| \{ r \in CC(m) \mid m \in M(con) \wedge r \notin CC(con) \} \right|$$

Table 3-6: Instantiation of new concern metrics for coupling and cohesion.

	Concern Coupling		Concern Cohesion	
	CSC	ICSC	LCC	CSCC
1. Entity	Concern	Concern	Component	Component
2. Property	[Efferent] Coupling	[Efferent] Coupling	Cohesion	Cohesion
3. Unit and Scale	Connections Interval	Connections Interval	Concerns Interval	None Ratio
4. Values	$[0, \sum^{c \in C(s)} CC(c)]$ Discrete	$[0, M(c)\text{-}M(con)]$ Discrete	$[0, Con(S)]$ Discrete	$[0, 1]$ Continuous
5. Granularity and Distinction	Component No	Member No	Component No	Member No
6. Domain, Inheritance	Application No	Application No	Application No	Application No
7. Concern, Artefact, and Overlapping	All Statement Yes	All Member Yes	All Component Yes	All Member Yes

Lack of Concern-based Cohesion (LCC) quantifies the cohesion of a *component* based on the number of *concerns* it realises. It counts the number of *concerns* realised by the target *component*. LCC is based on a similar metric defined for architecture models [133].

$$\begin{array}{lll} \mathrm{LCC}(c) = & \left| Con(c) \right| & \textit{if } c \in AC(S) \\ \mathrm{LCC}(c) = & 0 & \textit{otherwise} \end{array}$$

Concern-Sensitive Component Cohesion (CSCC) quantifies cohesion of a component based on the rate of *connections* among *operations, attributes,* and *declarations* assigned to a given concern. In other words, this metric is a result of the division of ICSC by the number of *members* of a component.

$$\begin{array}{lll} \mathrm{CSCC}(con, c) = & \dfrac{\mathrm{ICSC}}{\left| M(c) \right|} & \textit{if } c \in AC(S) \\ \mathrm{CSCC}(con, c) = & 0 & \textit{otherwise} \end{array}$$

Although these newly defined metrics (CSC, ICSC, LCC, and CSCC) quantify concern properties that are not covered by existing metrics, they were successfully instantiated in the concern measurement framework. This confirms the generality of the proposed framework to deal with new concern properties, such as coupling and cohesion. However, some minor extensions were required (Table 3-6). For instance, an annotation "efferent" was attached to the measurement properties of CSC and ICSC in order to show that these metrics count efferent connections, i.e., where components are playing the client role. This annotation is required because coupling connections have two directions: efferent (or fan-out) and afferent (or fan-in) [110].

3.5 Preliminary Framework Evaluation

Sections 3.3 and 3.4 described how to instantiate and formalise concern metrics using our conceptual framework. This section discusses to what extent the instantiated concern metrics are useful maintainability and stability indicators (Section 3.5.1). It also compares the concern metrics when they are applied to a software system (Section 3.5.2). In this section, our aim is to empirically verify the following hypothesis.

> ***Hypothesis 3-2****: Some existing concern metrics have overlapping measurement goals in the sense that they measure the same concern properties.*

3.5.1 Quantifying Concern Modularity and Design Stability

This section presents and discusses the results of applying 12 existing concern metrics in one of our case studies, called MobileMedia [56, 148]. Mobile Media (MM) is a software product line for handling data on mobile devices (see details in Section 7.1). We analyse the usefulness of these concern metrics to quantify modularity and stability of the MobileMedia system. We rely on a suite composed of all metrics formalised in Section 3.3, except Wong's metrics Concentration, Dedication, and Disparity [151]. These three metrics were excluded from our analysis because they are dynamic metrics [115, 151]. Therefore, they cannot be precisely quantified unless we have focused use case scenarios that exercise the execution of elements realising each target concern [151]. We also exclude the new concern metrics proposed in Section 3.4 because they do not contribute for testing the hypothesis of this section. The selected set of metrics quantifies four concern modularity properties (Section 4.2), namely scattering, tangling, closeness, and size. In addition to being successfully formalised in our framework, such metrics were chosen because they have already been used in several maintainability studies [41, 56, 61, 66, 73, 133].

We apply the suite of metrics to 12 concerns of MobileMedia which include 9 product-line features (Sorting, Favourites, SMS Transfer, Copy Media, Capture Media, Labelling, Photo, Music, and Video), 2 non-functional requirements (Exception Handling and Persistence), and the Chain of Responsibility (CoR) design pattern. The features were selected because they are the locus of variation in the software product line and, therefore, they have to be well modularised. On the other hand, non-functional requirements and the CoR design pattern are investigated because they are scattered in the design and tangled with core functionalities of the MobileMedia system. So, they may hinder the system design stability. Tables 3-7 and 3-8 show the results of applying 10 concern metrics (instantiated in Tables 3-2 and 3-3) to the selected set of concerns in the object-oriented version of MobileMedia. Two metrics, Concentration and Disparity, are going two be discussed in Section 3.5.2.

Table 3-7: Measurement of scattering and tangling for the OO MobileMedia.

		Scattering				Tangling
		CDC	CDO	Spread	FCD	CDLOC
DP	Chain of Resp.	14	16	14	-	36
NF	Exc. Handling	28	28	28	-	256
NF	Persistence	26	71	26	-	106
FT	Capture	8	18	8	8	16
FT	Copy	9	10	9	9	36
FT	Favourites	6	3	6	6	32
FT	Labelling	22	14	22	22	218
FT	Music	16	58	16	16	36
FT	Photo	12	50	12	12	26
FT	SMS Transfer	12	38	12	12	24
FT	Sorting	6	6	6	6	42
FT	Video	16	59	16	16	26

Table 3-8: Measurement of closeness and size for the OO MobileMedia.

		Closeness	Size			
		Focus	Size	NOCA	LOCC	Touch
DP	Chain of Resp.	29.46	17	1	554	3.89
NF	Exc. Handling	100.00	33	5	555	7.55
NF	Persistence	94.14	89	18	910	20.37
FT	Capture	84.36	38	20	297	8.70
FT	Copy	54.73	17	7	225	3.89
FT	Favourites	16.40	6	3	84	1.37
FT	Labelling	61.27	27	13	223	6.18
FT	Music	89.84	89	31	429	20.37
FT	Photo	89.82	72	22	445	16.48
FT	SMS Transfer	89.35	68	30	452	15.56
FT	Sorting	21.05	8	2	103	1.83
FT	Video	90.39	86	27	456	19.68

Tables 3-9 and 3-10 present similar data of the previous tables for the aspect-oriented version. The rows of Tables 3-7 to 3-10 list all concerns of the three different categories; the columns present the metrics. The first column identifies the category of each concern: features (FT), design patterns (DP), and non-functional requirements (NF). Only features were aspectised in MobileMedia because the aspectisation goal was to make them pluggable, i.e., features can be included in or excluded from a product-line instance. For all the employed concern metrics (except Focus [41]), a lower value implies a better result. From the analysis of the metrics, two distinct groups of concerns naturally emerged with respect to which solution – aspect-oriented (AO) or object-oriented (OO) – presents superior modularity properties.

Table 3-9: Measurement of scattering and tangling for the AO MobileMedia.

		Scattering				Tangling
		CDC	CDO	Spread	FCD	CDLOC
DP	Chain of Resp.	29	28	29	-	112
NF	Exc. Handling	34	47	34	-	194
NF	Persistence	35	90	35	-	106
FT	Capture	6	26	6	6	2
FT	Copy	13	17	13	13	14
FT	Favourites	7	9	7	7	4
FT	Labelling	34	14	34	34	252
FT	Music	15	43	15	15	6
FT	Photo	21	46	21	21	68
FT	SMS Transfer	14	54	14	14	6
FT	Sorting	6	11	6	6	4
FT	Video	14	40	14	14	4

Table 3-10: Measurement of closeness and size for the AO MobileMedia.

		Closeness	Size			
		Focus	Size	NOCA	LOCC	Touch
DP	Chain of Resp.	27.26	29	1	672	5.26
NF	Exc. Handling	100.00	52	5	611	9.44
NF	Persistence	94.17	108	18	951	19.60
FT	Capture	100.00	45	19	315	8.17
FT	Copy	100.00	23	6	211	4.17
FT	Favourites	100.00	13	4	99	2.36
FT	Labelling	63.77	27	13	295	4.90
FT	Music	100.00	70	27	462	12.70
FT	Photo	83.24	71	25	521	12.89
FT	SMS Transfer	100.00	87	33	471	15.79
FT	Sorting	100.00	14	3	127	2.54
FT	Video	94.69	58	18	408	10.53

Increased modularity of product-line features. The first group includes product-line features, such as Sorting and Favourites, which were separated into aspects in the aspect-oriented solution. Since these concerns were aspectised, they present reduced scattering and tangling compared to the object-oriented counterpart for almost every metric, except CDO. The CDO value increased in the aspect-oriented solution because pieces of code tangled inside a method in the object-oriented design were moved to newly created advice (accounted as operations for CDO). The developers' intention is also closer materialised in the design structural elements (closeness) of the aspect-oriented solution according to the Focus metric. Interestingly, however, the aspectisation of features does not reduce the concern size as highlighted by the measurement in Tables 3-8 and 3-10. In fact, most of the features increased their size with the aspectisation process. This problem arises because the AspectJ implementation faces difficulties to address different product-line configurations (i.e., specific combination of features) [56]. In particular, in the aspect-oriented implementation of MobileMedia there is barely any gain in reuse of features-related code. Therefore, the overhead of AspectJ constructs (e.g., new pointcuts and declarations) overcomes the potential benefits in terms of reuse in this application.

Decreased modularity of crosscutting concerns. The second group of analysed concerns includes Persistence, Exception Handling, and the CoR design pattern. Although those concerns crosscut the core functionalities of MobileMedia, they were not modularised into aspects (the developers' strategy was to modularise only varying features). As a result, Tables 3-7 to 3-10 show that their respective metrics for all concern properties have higher values in the AO solution. The inferior modularity of these concerns in the AO solution is expected since most features depend on the crosscutting concerns. For instance, as new aspects are introduced to separate optional and alternative features of the product line, crosscutting concerns are spread over those newly introduced components. Furthermore, the overhead of the AspectJ implementation contributes to the degradation of non-modularised crosscutting concerns. Therefore, the analysed crosscutting concerns are less close to the intention of structural elements and present more scattering, tangling, and size in the AO solution.

3.5.2 Comparison of Concern Metrics

In this section we compare our representative set of concern metrics using two strategies: their formalisation in our framework (Section 3.3) and their results obtained from MobileMedia (Section 3.5.1). As observed before (Sections 2.2 and 3.3), the concern metrics CDC, CDA, Spread and FCD have similar definitions. Actually, according to their formal definitions in the measurement framework, CDC and CDA have exactly the same instantiation (see Table 3-2 in Section 3.3.2). Based on this observation, the hypothesis of this section is verified. That is, some existing concern metrics measure the same concern property.

In addition to confirm that similarity, the framework also helped to point out small differences among the concern metrics. For instance, while CDC and Spread (Table 3-3) are applied to all sorts of concerns, the FCD metric (Table 3-2) is only intended for features [102]. Furthermore, overlapping of concerns is permitted in CDC and FCD, but it is not allowed in the definition of Spread due to constraints in the concern representation (Distribution Map [41]). Moreover, the two concern metrics Concentration and Dedication are both defined by Wong, Gokhale, and Horgan [151] and Eaddy, Aho, and Murphy [46]. In spite of their similarities, the formalisation of both suites of metrics reveals that the granularity is members in the former and lines of code in the latter (Table 3-4).

In addition to compare the concern metrics based on their formal definitions, the results of our analysis in the MobileMedia system not only confirmed some observations but also revealed further findings. For instance, the similarities among CDC, CDA, Spread and FCD were confirmed by their equivalent values in the MobileMedia study (Section 3.5.1). On the other hand, from our empirical analysis we learned three additional correlations among the selected concern metrics. First, we observed that the Size metric is always a summation of the CDO and NOCA metrics. Second, the Touch metric is directly proportional to Size when they are applied to the same system, i.e., MobileMedia in our study. Third, for a given concern, the Concentration metric from Eaddy, Aho, and Murphy [46] is proportional to the LOCC metric. The first two observations above can be verified in Tables 3-8 and 3-10.

To illustrate that Eaddy's Concentration metric is proportional to his LOCC metric for a given concern (the third observation above), we rely on data of Tables 3-11 and 3-12. Table 3-11 shows the results of the metrics LOCC, Concentration, and Dedication for the Sorting concern in the object-oriented design of MobileMedia. Table 3-12 presents the results of the same metrics applied to the corresponding aspect-oriented design. It is easy to observe that the value of Concentration in Table 3-11 increases as more lines of code are used in the concern realisation. Furthermore, Table 3-12 indicates that every line of concern code (LOCC) accounts for almost 1 % in the Concentration metric in this specific system.

Table 3-11: Metrics of Eaddy *et al*. [46] applied to the OO implementation of Sorting.

	LOCC	Concentration	Dedication
ImageData	14	14.58	25.00
ImageUtil	20	20.83	17.70
PhotoController	31	32.29	13.48
PhotoListController	25	26.04	39.06
PhotoListScreen	6	6.25	26.09

Table 3-12: Metrics of Eaddy *et al*. [46] applied to the AO implementation of Sorting.

	LOCC	Concentration	Dedication
SortingAspect	102	99.03	100.00
OptionalFeaturesAspect	1	0.97	33.33

3.6 Summary

In this chapter we have presented a conceptual framework composed of terminology and formalism for concern analysis (Section 3.1). We also provided a set of criteria for instantiating and comparing the definition of concern metrics (Section 3.2). Section 3.3 provided detailed guidance in order to define concern metrics in a consistent and operational way. It also defined and compared a representative set of concern metrics using the framework's criteria (Section 3.5).

By analysing the state-of-the-art, we verified that there is a very rich body of ideas regarding the way concern measurement is addressed. However, we verify that some concerns' modularity properties, such as coupling and cohesion, are not addressed by existing concern metrics (hypothesis of Section 3.3). To fill this gap, new concern metrics were proposed in Section 3.4 to quantify coupling and cohesion of components based on the concern realisation. Furthermore, we confirmed that many existing concern metrics have similar definitions and measurement goals (hypothesis of Section 3.5). This finding is demonstrated by the instantiation of several concern metrics in the proposed framework (Section 3.3) and their comparison in the context of an empirical study (Section 3.5).

The terminology and formalism proposed in this chapter is intended to be abstract and language-independent since the framework is supposed to be extensible. However, by no means have we claimed that these proposed definitions are exhaustive; further extensions might be necessary. Some of these extensions were briefly discussed in Section 3.4. Although this chapter focused on concern measurement, Chapters 4 and 5 rely on the same formalism presented here for higher-level concern analysis. For instance, in the next chapter the proposed formalism is used to define thirteen patterns of crosscutting concerns.

4. Patterns of Crosscutting Concerns

As introduced in Chapter 2, the inappropriate modularisation of crosscutting concerns, such as exception handling and distribution, may lead to undesirable software instabilities [44, 129, 136]. For instance, concern scattering tends to both destabilise software modularity properties and increase the propagation of changes according to empirical assessments [44, 56, 73]. There is also initial evidence that not all types of crosscutting concerns are harmful to the system stability [56, 109]. However, there is not much understanding on what specific types of crosscutting concerns are likely to deteriorate design stability. In fact, the realisation of crosscutting concerns affects modularity units of a system in significantly-different forms, ranging from a few elements in a specific inheritance tree to code fragments scattered across the entire system. Hence, stability analysis of crosscutting concerns can not be carried out without the systematic classification of these concern's realisation forms.

Chapter 2 also discussed some recent studies [41, 79, 108, 109] which have proposed ways of classifying crosscutting concerns (Section 2.4). Most of them focus on refactoring for modularising specific types of crosscutting concerns. For example, role-based refactoring [79] and refactoring of concern sorts [108] are two different approaches with this aim. Although these categories of concerns could be seen as preliminary steps towards the definition of a catalogue of crosscutting concerns, these approaches suffer from some limitations. Firstly, descriptions of concerns in these refactoring approaches are not abstract and generic enough to be considered in different contexts. In other words, they cannot be applied to design artefacts or beyond the context of specific programming language mechanisms. For example, names for roles [79] are based on a very restricted set of specific low-level concerns (e.g., roles defined by design patterns). In addition, concern sorts [108, 109] are usually tied to constructs of specific AOP languages (e.g., *exception softening* mechanisms of AspectJ).

The previous chapter proposed a generic terminology for concern analysis. It also defined the foundations for identifying and analysing using metrics some properties of concerns, such as scattering, tangling, and coupling. This chapter relies on the previously defined terminology and formalism (Section 3.1) to present a systematic documentation of thirteen typical patterns of crosscutting concerns, i.e., *crosscutting* *patterns*. As the formalisation is agnostic to language intricacies, the proposed crosscutting patterns can be used to anticipate instabilities in both implementation-level and design-level artefacts. Section 4.1 briefly describes how the crosscutting patterns have been defined. The crosscutting patterns are classified into four categories: Flat Crosscutting Patterns (Section 4.2), Inheritance-wise Concerns (Section 4.3), Communicative Concerns (Section 4.4), and Other Crosscutting Patterns (Section 4.5). Section 4.6 discusses how the proposed crosscutting patterns correlate to each other and Section 4.7 summarises this chapter.

4.1 Defining Crosscutting Patterns

The catalogue of crosscutting patterns to be presented in this chapter is inspired by the work from Ducasse, Girba, and Kuhn [41]. As introduced in Section 2.4, these authors identified four patterns of concerns: Well-Encapsulated, Crosscutting, Octopus, and Black Sheep. They rely on a set of concern metrics and in a visualisation technique to characterise these patterns in software systems. Following this seminal work, we have been observing crosscutting patterns in several software systems. Our experience in analysing crosscutting concerns is a result of empirical studies involving the following systems: MobileMedia [52, 54], Health Watcher [73, 137], a Design Pattern library [80], OpenOrb Middleware [22], AJATO [1, 58], Eclipse CVS Plugin [22, 36], and Portalware [68, 120]. These systems were used because they (i) come from different domains, (ii) offer varying degrees of complexity, and (iii) involve heterogeneous forms of crosscutting concerns.

Based on this experience, we provide a catalogue of crosscutting patterns which facilitates their understanding and investigation in both object-oriented and aspect-oriented systems. This catalogue is based on characteristics of the concern realisation, such as coupling and inheritance relationships, which are likely to impact on software stability. In our catalogue, each crosscutting pattern is defined by a textual description, a formal definition, and an example. We rely on the formalism presented in Section 3.1 to provide the patterns' formal definitions. As far as the examples are concerned, we use an abstract pictorial representation and may also provide concrete pattern instances. The concrete examples are mainly extracted from Health Watcher which is one of our case studies (Chapter 8). Health Watcher is a Web-based information system for supporting healthcare-related complaints [73, 137]. The next sections present the proposed catalogue of crosscutting patterns. A concern can be classified in one (or more) crosscutting pattern, provided that it is first identified as crosscutting. We discuss how to identify crosscutting concerns in Section 5.1.

4.2 Flat Crosscutting Patterns

This section describes and formalises three patterns of crosscutting concerns, namely *Black Sheep*, *Octopus*, and *God Concern*. The first two patterns (Octopus and Black Sheep) have already been identified in previous work by Ducasse, Girba, and Kuhn [41] (Section 2.4), although they have never been formalised and used in concern analysis of design stability. God Concern and the other 10 crosscutting patterns of this chapter represent our original contribution towards a unified catalogue.

4.2.1 Black Sheep

The *Black Sheep* crosscutting pattern (also called *Lazy Concern*) is defined as a concern that crosscuts the system but is realised by very few elements in distinct components [41]. This crosscutting pattern can also be called "Lazy Concern" because it typically involves very few design elements in the concern realisation. For instance, Black Sheep occurs when only a few scattered operations and attributes are required to realise a concern in design artefacts. Also, there is not a single component whose main purpose is to implement this concern. Figure 4-1 presents an abstract representation of a Black Sheep concern. The shadow grey areas in this figure indicate elements of the components (represented by boxes) realising the concern under consideration. Taking these six components into account, the Black Sheep instance is realised by few elements of two components.

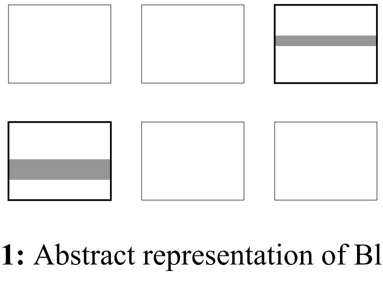


Figure 4-1: Abstract representation of Black Sheep

Relying on the formalism introduced in Section 3.1, we present a formal definition of the Black Sheep crosscutting pattern. Consider *c* a component in *C(S)* and *con* a concern in *Con(S)*. As defined earlier (Section 3.1), *C(S)* consists of the set of components of a system *S* and *Con(S)* represents the set of concerns addressed by the system *S*. We also define the *dedication(c, con)* relation that indicates how much of the component *c* is used to realise the concern *con*. We discuss how to measure dedication of a component to a concern in Section 5.1. Based on this relation, we define a concern *con* to be Black Sheep as follows.

$$\text{BlackSheep(con)} := \text{C(con)}$$

If and only if (condition):

$$\forall\ \text{c} \in \text{C(con) dedication(c, con)} < 33\ \%$$

As stated above, when the condition holds we define that all components in *C(con)* are composing the Black Sheep crosscutting pattern. This definition is also adopted for all crosscutting patterns of this chapter. That is, the set which defines a crosscutting pattern includes all components realising the concern, provided that the pattern condition holds. However, for convenience, we omit the first part of the formal definition not only for Black Sheep but also for the other crosscutting patterns of this chapter. In this case, the simplified formal definition of Black Sheep is expressed as follows.

$$\text{BlackSheep(con)} \Leftrightarrow\ \forall\ \text{c} \in \text{C(con) dedication(c, con)} < 33\ \%$$

This expression says that all components realising a Black Sheep concern dedicate less than 33% of their data and behaviour to this concern. The choice of 33% is derived from general evidence-based guidelines for selecting thresholds [95], and it could obviously be adapted according to application-specific contexts and the designer's experience. However, Lanza and Marinescu [95] suggest the use of meaningful threshold values, such as 33% (1/3), 50% (1/2), and 67% (2/3). Based on this recommendation, we use these percentages meaning Low (33%), Moderate (50%), and High (67%)[3]. In fact, our tool presented in Section 5.6 supports the definition of thresholds values customised by users.

An example of Black Sheep is the Singleton design pattern [64] presented in the partial design of Figure 4-2. This Singleton solution matches the Black Sheep crosscutting pattern because the `HealthWatcherSingleton` class requires only one

[3] The same rationale applies to the other thresholds in this chapter.

attribute (`instance`) and one operation (the `getInstance()` method) to realise the pattern's concern. Moreover, this concern is crosscutting because there are many calls to the `getInstance()` method spread over other classes in the system (not shown in Figure 4-2).

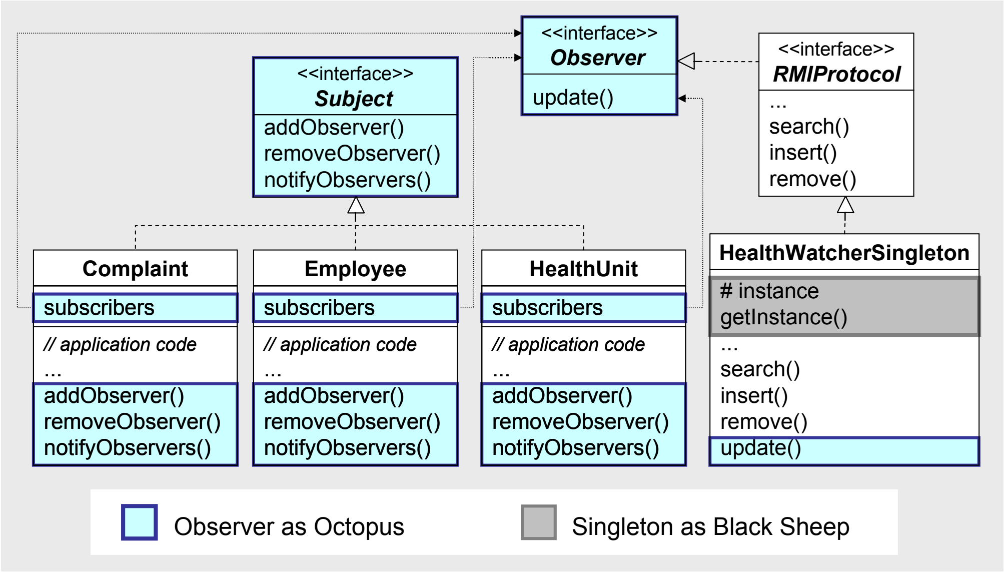


Figure 4-2: Instances of flat crosscutting patterns in the Health Watcher design slice.

4.2.2 Octopus

The second crosscutting pattern in this group, *Octopus*, is defined as a crosscutting concern which is partially well modularised by one or more components, but it is also spread across a number of other components [41]. Figure 4-3 presents an illustrative abstract representation of the Octopus crosscutting pattern. Note that two components (boxes) in this figure are completely dedicated to the concern realisation (they represent the *octopus' body*), while four other components (representing *tentacles*) dedicate only small parts to realise the same concern.

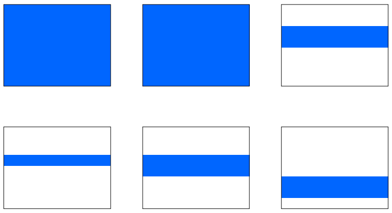


Figure 4-3: Abstract representation of Octopus

We use the previously defined *dedication(c, con)* relation to give the following formal definition of Octopus. As stated in the expression below, a concern is classified as Octopus if, and only if, at least one component *b* dedicates more than 67% and another component *t* dedicates less than 33% to the concern realisation. Unlike Black Sheep, the Octopus crosscutting pattern does not require that all components realising the concern match specific constraints (note the *existential* quantification instead of the *universal* one). In other words, even if some components do not take the part of tentacles or body (e.g., they dedicate between 33% and 67% of their functionality to the concern realisation), the concern may still be classified as Octopus. It is important to emphasise that the Octopus set includes all components realising the concern regardless of their degree of dedication. That is, *Octopus(con)* is equal to *C(con)* if the condition holds. All crosscutting patterns follow this definition.

$$\text{Octopus(con)} \Leftrightarrow \exists\ \text{b}\ \exists\ \text{t} \in \text{C(con) dedication(b, con)} > 67\ \%$$
$$\wedge\ \text{dedication(t, con)} < 33\ \%$$

Figure 4-2 presents an instance of the Observer design pattern [64] which matches the definition of Octopus. This Octopus instance has two interfaces, `Subject` and `Observer`, completely dedicated to the concern realisation (the octopus' body) and four classes with only few methods and attributes realising Observer (the octopus' tentacles). We should emphasise that the number of components playing the role of body and tentacles in the Octopus pattern is not fixed. Therefore, provided that there are components playing both body and tentacles roles, this crosscutting pattern is characterised.

4.2.3 God Concern

Inspired by the God Class bad smell [63], the *God Concern* crosscutting pattern occurs when a concern encompasses too much scattered functionality of the system. In other words, similarly to God Class, the elements realising God Concern address too much or do too much across the software system. The basic idea behind modular programming is that a widely-scoped concern should be analysed as smaller "modular" sub-parts (*divide and conquer* [91]). However, God Concern instances do not adhere to this principle; thus, hindering concern analysis.

Figure 4-4 presents the abstract structure of this crosscutting pattern. This figure highlights that God Concern is a widely-scoped crosscutting concern and requires a lot of functionality in its realisation. In particular, there are at least three components whose main purpose is to realise this concern. On the other hand, Figure 4-4 also shows that at least one component affected by a God Concern instance does not have the main purpose of realising it. That is, this concern must crosscut at least one component of the system to be considered a crosscutting pattern. We define that a concern is crosscutting a component when the dedication of this component to the concern realisation is low (we elaborate more on this in Section 5.1).

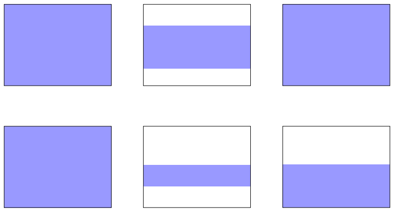


Figure 4-4: Abstract representation of God Concern

The formalisation of God Concern considers the overall dedication of the system's components to the given concern. Hence, it requires our *dedication()* relation to be overridden as given below. The *dedication(C(S), con)* relation defines the average dedication of components in *C(S)* to the concern *con*. Then, to be considered God Concern, *dedication(C(S), con)* needs to be higher than 33%. In other words, at least 33% of the overall system functionality is assigned to a single concern. Furthermore, there exists at least one component $c \in C(con)$ which dedicates less than 50% of its functionality to the concern realisation. This last condition aims to indicate that the analysed concern is crosscutting at least one component.

$$\text{GodConcern(con)} \Leftrightarrow \exists\ c \in \text{C(con) dedication(c, con)} < 50\% \\ \wedge\ \text{dedication(C(S), con)} > 33\%$$

Where,

$$\text{dedication(C(S), con)} = \frac{\sum^{c \in C(S)} \text{dedication(c, con)}}{|\ C(S)\ |}$$

An example of God Concern is the Business concern which is found in Health Watcher [72, 73] (Section 8.1). Business is an architectural layer as presented in Figure 8-2 (Chapter 8). It is considered as a global concern of major interest in this application and, as such, its sub-parts should be also modularised in later design artefacts. This concern encompasses different sub-concerns, such as Complaint, Disease, Employee, and Health Unit (among others). Although all these concerns are related to the business logic, a concern analysis should consider its several sub-concerns since each of them represents a different modular slice of the Business rules.

4.3 Inheritance-wise Concerns

This section presents the second group of crosscutting patterns. This group includes two patterns, namely *Climbing Plant* and *Hereditary Disease*, which manifest themselves in inheritance trees. That is, we consider inheritance relationships in the definitions of these crosscutting patterns.

4.3.1 Climbing Plant

Climbing Plant is a pattern to identify crosscutting concerns that affect the *root* of an inheritance tree and their structure is propagated to all children in this tree. This crosscutting structure can be totally or partially propagated to the root descendants. However, all descendants (also called *branches*) of the concern root are somehow affected. Additionally, it is not required for all components realising the Climbing Plant concern to be branches or the root. For instance, Figure 4-5 presents a representation of Climbing Plant. Two components of this figure are not taking part in the inheritance tree, although they realise the concern and so the crosscutting pattern. For the other four components, they correspond to all elements inheriting from a common root component. The main problem caused by concerns matching this crosscutting pattern is related to implicit dependencies involving their forming components. These dependencies become clear when a change targeting a concern branch ripples through the root to other branches.

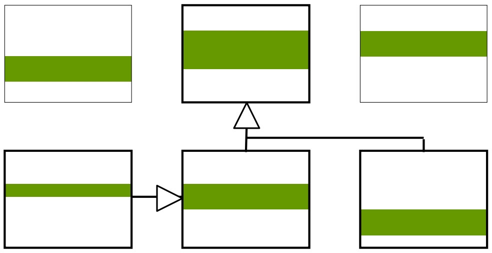


Figure 4-5: Abstract representation of Climbing Plant

To formalise Climbing Plant, an additional transitive relation, *descendant(c1, c2)*, is required. This relation returns true if *c1* $\in$ *C(S)* directly or indirectly inherits from *c2* $\in$ *C(S)*. Then, we define three sets of components, *Root*, *Branches* and *DiseaseFreeNodes*, as presented below. The first set, *Root(con)*, is a subset of *C(con)* which includes all root components. A root component *r* is a component in *Root(con)* which does not inherit from any other component in *C(con)*, although it may inherit from a component in *C(S)*. The *Branches(con)* set includes all components realising the concern *con* and inheriting from a root component. Components in the *DiseaseFreeNodes(con)* set do not realise the concern *con*, although they also inherit from a root component. The two sets *Branches* and *DiseaseFreeNodes* are also used in the formalisation of the next crosscutting pattern (Section 4.3.2). Finally, a concern is classified as Climbing Plant if, and only if, the *Branches* set is not empty and the *DiseaseFreeNodes* set is empty.

$$\text{ClimbingPlant(con)} \Leftrightarrow \text{Branches(con)} \neq \emptyset \wedge \text{DiseaseFreeNodes(con)} = \emptyset$$

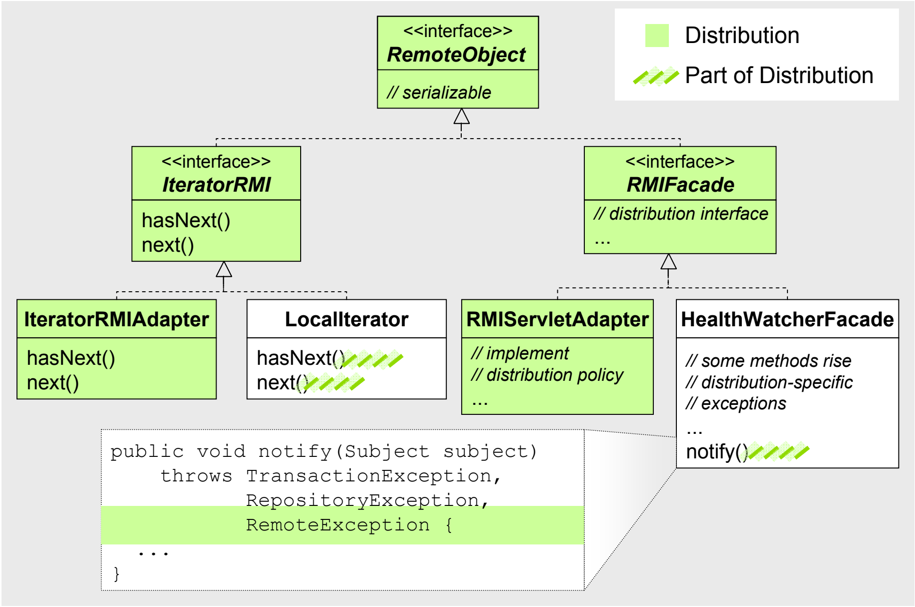
Where,

$$\text{Root(con)} := \{ r \in \text{C(con)} \mid \nexists\ c \in \text{C(con)}\ \text{descendant}(r, c) \wedge c \neq r \}$$

$$\text{Branches(con)} := \{ c \in \text{C(con)} \mid \exists\ r \in \text{Root(con)}\ \text{descendant}(c, r) \wedge c \neq r \}$$

$$\text{DiseaseFreeNodes(con)} := \{ c \in \text{C(S)} \mid \exists\ r \in \text{Root(con)}\ \text{descendant}(c, r) \wedge c \notin \text{C(con)} \}$$

Figure 4-6 shows some classes realising the Distribution concern in the Health Watcher system [73, 137]. Note that, all classes in this partial design realise the Distribution concern, although some of them (i.e., `HealthWatcherFacade` and `LocalIterator`) just dedicate a small part of their functionalities to the concern realisation. Since these classes are also organised in a tree via inheritance relationships, Distribution is classified as Climbing Plant in this system. The `RemoteObject` interface is the root component of this pattern instance.



```
public void notify(Subject subject)
    throws TransactionException,
           RepositoryException,
           RemoteException {
  ...
}
```

Figure 4-6: Distribution as Climbing Plant

4.3.2 Hereditary Disease

As with Climbing Plant, *Hereditary Disease* also affects the root of an inheritance tree and, then, propagates its crosscutting structure to *some* root descendants. However, a concern matching the Hereditary Disease crosscutting pattern does not manifest itself in all nodes of a tree, i.e., some nodes are *disease-free*. Figure 4-7 illustrates the Hereditary Disease structure and, together with Figure 4-5, we can see the key difference between these two crosscutting patterns (Climbing Plant and Hereditary Disease). In particular, there is a disease-free node in the Hereditary Disease example – marked using the red colour in Figure 4-7. Disease-free nodes are not allowed in the Climbing Plant definition. Apart from that, the modularity problems are similar and require analogous solutions.

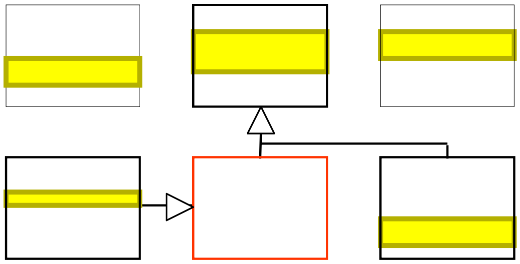


Figure 4-7: Abstract representation of Hereditary Disease

A concrete instance of Hereditary Disease can be noticed in the Observer design pattern presented in Figure 4-2. All classes and interfaces in Figure 4-2 realise the Observer concern, except the `RMIProtocol` interface. Also, the components are organised in an inheritance tree rooted at the `Subject` and `Observer` interfaces. Hence, they form the Hereditary Disease crosscutting pattern with the `RMIProtocol` interface representing a disease-free node. The pattern formal definition relies on the two sets of components, *Branches* and *DiseaseFreeNodes*, defined before for Climbing Plant. Then, a concern is classified as Hereditary Disease if, and only if, both sets of components are not empty.

$$\begin{aligned}\text{HereditaryDisease(con)} \Leftrightarrow\ & \text{Branches(con)} \neq \emptyset \\ & \wedge \text{DiseaseFreeNodes(con)} \neq \emptyset\end{aligned}$$

4.4 Communicative Concerns

This section presents four patterns of crosscutting concerns: *Tree Root*, *Tsunami*, *King Snake*, and *Neural Network*. The particular characteristic of components realising concerns of this pattern category is that they have coupling connections among them. The different ways in which components connect to each other define the distinct crosscutting patterns.

4.4.1 Tree Root

The *Tree Root* crosscutting pattern is formed by two kinds of components: a *trunk* and *feeders*. The trunk of Tree Root is formed by components that only receive incoming connections from other components. The feeders, on the other hand, represent components which directly or indirectly connect to a trunk component. Direct connections can be, for instance, method calls or attribute accesses (see Section 3.1.3). Additionally, more than one component can play the role of trunk. However, several feeders must be connected to just a few trunk components in this crosscutting pattern. The reasoning is that trunk components may represent a small spot which is the source of design instability since many feeders rely on these few components. Moreover, in a typical situation, there is just a single trunk component as shown in the abstract representation of Tree Root presented in Figure 4-8. This figure shows four feeders connected to a trunk component. The trunk component is represented by a box with brown borders in this figure. Not all components realising a Tree Root concern are required to be connected to the trunk, e.g., Figure 4-8 shows one such component.

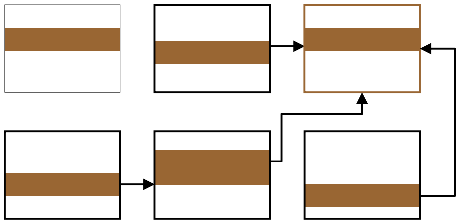


Figure 4-8: Abstract representation of Tree Root

For a formal definition of Tree Root, we define the non-reflexive *connected(c1, c2)* relation that maps to a Boolean value indicating if a component *c1* connects to a component *c2* (both *c1* and *c2* realising the concern *con* and *c1* is different of *c2*). This relation is transitive, i.e., given *c1*, *c2* and *c3* $\in$ *C(con),* if *connected(c1, c2)* and *connected(c2, c3)* then *connected(c1, c3)*. We also define below the *Trunk* and *Feeders* set. According to these expressions, the *Trunk* set is composed of components realising a concern *con* and these components do not connect to any other component in *C(con)*. Similarly, *Feeders* is a subset of *C(con)* so that each feeder component is connected to at least one component in the *Trunk* set. We define a concern to be Tree Root if, and only if, there is at least one component in the *Trunk* set. Moreover, the *Feeders* set must be larger than the *Trunk* one.

$$\text{TreeRoot(con)} \Leftrightarrow (\mid \text{Feeders(con)} \mid > \mid \text{Trunk(con)} \mid) \wedge \text{Trunk(con)} \neq \emptyset$$

Where,

$$\text{Trunk(con)} := \{ t \in \text{C(con)} \mid \nexists\ c \in \text{C(con) connected(t, c)} \}$$

$$\text{Feeders(con)} := \{ f \in \text{C(con)} \mid \exists\ t \in \text{Trunk(con) connected(f, t)} \wedge f \neq t \}$$

Figure 4-9 provides a concrete instance of Tree Root extracted from the Health Watcher system [73, 137]. This figure shows some components realising the State design pattern [64] which are organised according to the Tree Root pattern. Some details of State, such as inheritance relationships, are hidden for simplification purposes. In this Tree Root instance, the `Situation` class is playing the role of trunk and eight components are connected to this trunk class. The design pattern structure allows these classes accessing the attributes (`openedComplaint`, `suspended-`

`Complaint`, and `closedComplaint`) of `Situation` in order to define the current status/state of each complaint class.

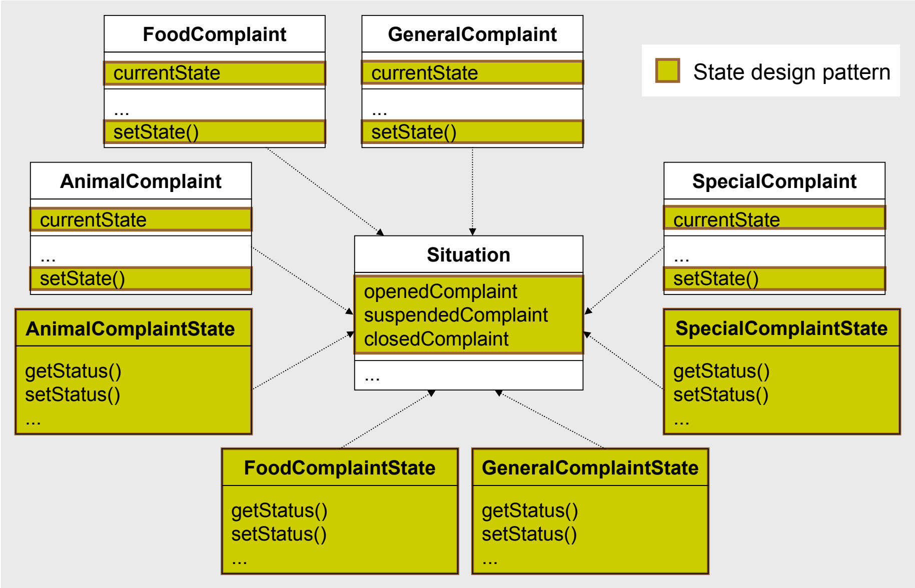


Figure 4-9: The State design pattern as Tree Root

4.4.2 Tsunami

The *Tsunami* crosscutting pattern is formed by a small set of components, named the *wave source*, which directly or indirectly connects to other components realising the same concern. This pattern resembles wave propagation through coupling connections. Unlike Tree Root, connections do not converge to a small set of components in the Tsunami pattern. In fact, Tsunami has the inverse configuration in comparison to Tree Root. That is, a few components (the wave source) rely on many others. Therefore, the wave source may represent a small spot which is the target of changes due to several reasons. In other words, the wave source includes a few components which depend on many others and, therefore, may be highly unstable. In a typical situation, the wave source is represented by a single component. The extremes in the “channel” of called components are named *wave limits* (i.e., those components realising the concern that just receive connections).

Components playing the role of the wave source do not need to be connected to every component realising the Tsunami concern. For instance, Figure 4-10 shows the overall Tsunami structure where one component is not being called by any other pattern component. The box with blue borders represents a component playing the role of the wave source. Comparing Figures 4-8 to 4-10, we verify the key difference between the Tree Root and Tsunami crosscutting patterns. That is, the directions of connections are converging to the trunk in the former and diverging from the wave source in the latter.

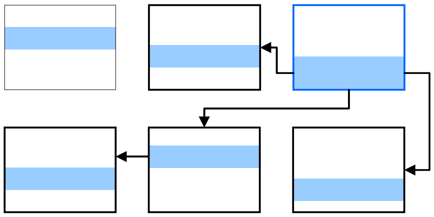


Figure 4-10: Abstract representation of Tsunami

The Tsunami formal definition (below) is also similar to the Tree Root one. It states that *Source* and *Waves* are a subset of *C(con)*, i.e., they include components which realise a concern *con*. Furthermore, there are components in the *Source* set which connects to every component in the *Waves* set. A concern is then classified as Tsunami if, and only if, the *Waves* set is larger than the *Source* set. Moreover, the *Source* set must not be empty.

$$\text{Tsunami(con)} \Leftrightarrow (\,|\,\text{Waves(con)}\,| > |\,\text{Source(con)}\,|\,)\ \text{Source(con)} \neq \varnothing$$

Where,

$$\text{Source(con)} := \{\ s \in \text{C(con)} \mid \nexists\ c \in \text{C(con) connected(c, s)}\ \}$$

$$\text{Waves(con)} := \{\ w \in \text{C(con)} \mid \exists\ s \in \text{Source(con) connected(s, w)} \wedge s \neq w\ \}$$

Figure 4-11 shows some classes realising the Persistence concern in Health Watcher. These classes are organised according to the Tsunami crosscutting pattern. The wave source of this pattern instance is represented by the `HealthWatcherFacade` class which directly or indirectly calls ten other classes. Therefore, if this class is executed, ten other classes may also be potentially called due to the propagation of coupling connections among them. Moreover, if one of the ten classes requires changes due to maintenance activities, the wave source (`HealthWatcherFacade` in this case) is also likely to be updated.

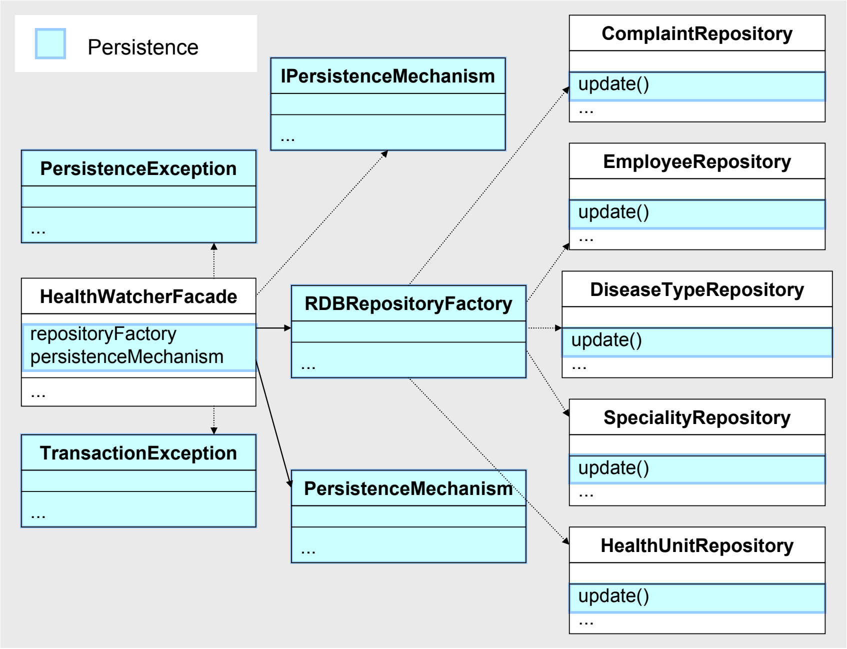


Figure 4-11: Persistence as Tsunami

4.4.3 King Snake

The third crosscutting pattern in the group of Communicative Concerns, *King Snake*, is characterised by a large chain of connected components realising the given concern. The first component in the chain is named *head* of the snake and the final connected component (end of the chain) is named *tail*. More than one component can play each of the head and tail roles resulting in more than one chain. In cases where a concern has more than one chain of components in its structure, we consider the longest chain to represent the King Snake instance. This information is important, for instance, when someone wants to compare King Snake instances (of different concerns) based on the longest component chain. The definition of this crosscutting pattern allows that some components realising the concern do not participate in the chain of connected components.

Figure 4-12 illustrates the abstract representation of King Snake. In this figure, four components are connected to one another due to the concern under consideration. However, there are also two components realising the concern which are not taking part in this chain. The component with light green borders represents the head of the snake while the component with dark green borders represents its tail. Alternatively, someone may call tail all components forming the snake, except the head one.

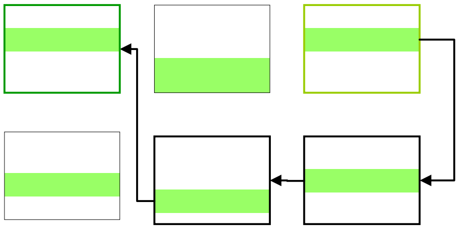


Figure 4-12: Abstract representation of King Snake

To formalise King Snake, we define *Chain* as a set of components realising a concern *con* and participating in the chain that connects these components. We also use the previously defined *Trunk* and *Source* sets (Sections 4.4.1 and 4.4.2, respectively). However, these sets are renamed *Head* and *Tail* (below) in order to rely on the intuitive use of snake-related terminology. The *Chain* set includes components in *C(con)* which exhibit two properties. First, components in *Head(con)* are connected to all components in *Chain(con)*. Second, every component in *Chain(con)* is connected to at least one component in *Tail(con)*. We must remember that the *connected()* relation is transitive.

$$\text{KingSnake(con)} \Leftrightarrow \text{Chain(con)} \neq \varnothing$$

Where,

$$\text{Head(con)} := \text{Source(con)}$$

$$\text{Tail(con)} := \text{Trunk(con)}$$

$$\text{Chain(con)} := \{\, c \in \text{C(con)} \mid \exists\ h \in \text{Head(con)}\ \text{connected}(h, c) \wedge \exists\ t \in \text{Tail(con)}\ \text{connected}(c, t) \wedge h \neq c \neq t \,\}$$

Figure 4-13 shows a concrete instance of King Snake. In this figure, we use both a static UML class diagram and a dynamic UML sequence diagram of a partial design of Health Watcher. This partial design discloses four classes realising (part of) the Abstract Factory design pattern [64]. These classes are organised as a chain. In particular, the `HealthWatcherFacade` class is playing the *head* role of this King Snake instance while `AbstractRepositoryFactory` is playing the role of the King Snake *tail*.

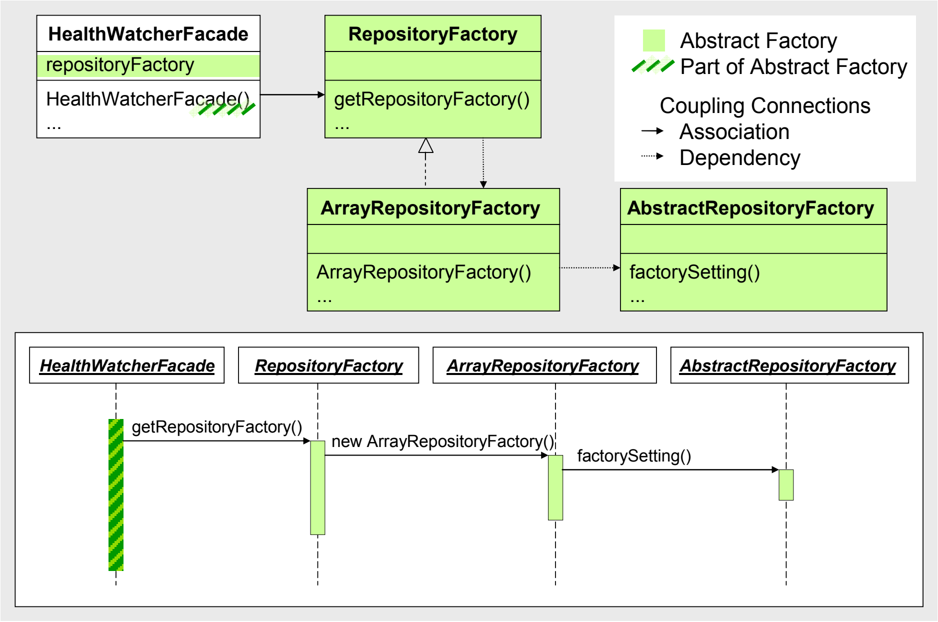


Figure 4-13: Abstract Factory as King Snake

4.4.4 Neural Network

The last crosscutting pattern in this group, *Neural Network*, consists of a large net of inter-connected components. Each component in the net is connected to one or more other components as presented in Figure 4-14. The overall configuration of this pattern resembles an artificial neural network including (i) components which are not called by any other component (*input layer*); (ii) components which do not connect to any other component (*output layer*); (iii) and optional components at the intermediary structure receiving and making connections (*hidden layers*). The abstract representation of Figure 4-14 shows two components in the *input layer* (with dark green borders), two in the *output layer* (with light green borders), and one in the *hidden layer*. Moreover, there is also one component realising the concern which is not connected to any other component.

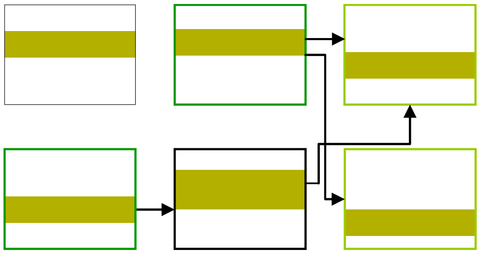


Figure 4-14: Abstract representation of Neural Network

The formal definition of Neural Network relies on two sets of components, called *InputLayer* and *OutputLayer* (defined below). In fact, these sets were formalised before as *Source* in Section 4.4.2 and *Trunk* in Section 4.4.1, respectively. The key properties of components in these sets are: (i) there is no component in the *C(con)* set connected to components in the *InputLayer* set and (ii) components in the *OutputLayer* set are not connected to any other component realising the same concern.

$$\begin{aligned}\text{NeuralNetwork(con)} \Leftrightarrow\ & (\ \exists\ i \in \text{InputLayer(con)}\ \exists\ o\ \exists\ p \in \text{OutputLayer(con)} \\ & \quad \text{connected}(i, o) \wedge \text{connected}(i, p) \wedge o \neq p\ ) \vee \\ & (\ \exists\ i\ \exists\ j \in \text{InputLayer(con)}\ \exists\ o \in \text{OutputLayer(con)} \\ & \quad \text{connected}(i, o) \wedge \text{connected}(j, o) \wedge i \neq j\ )\end{aligned}$$

Where,

InputLayer(con) := Source(con)
OutputLayer(con) := Trunk(con)

According to the formal definition above, a concern is classified as Neural Network if, and only if, there are at least two connecting paths from components in the input layer to components in the output layer. Moreover, either the input layer or the output layer must have at least two components. Then, considering the input and output layers, at least one component in one layer must be connected to two or more different components in the other layer.

Considering associations and inheritance relationships as connections[4], Figure 4-2 shows an instance of Neural Network represented by components of the Observer design pattern. In this Neural Network instance, at least three components are composing the input layer, namely `Complaint`, `Employee`, and `HealthUnit`. Moreover, two other components, `Subject` and `Observer`, are composing the output layer. Components in the input layer are connected to the `Subject` interface via inheritance relationships and to the `Observer` interface via associations.

[4] This definition is not fixed in our formalism. For instance, someone may consider just associations as connections when detecting crosscutting patterns in the category of Communicative Concerns.

4.5 Other Crosscutting Patterns

This group includes four patterns of crosscutting concerns, namely *Copy Cat*, *Dolly Sheep*, *Data Concern*, and *Behavioural Concern*, which do not fit in the previous categories. These crosscutting patterns express either replicated code or misbalance of data and behaviour in the concern realisation.

4.5.1 Copy Cat

Previous work [104] suggests that having a code clone may increase the maintenance effort of changing the system. *Copy Cat* is a crosscutting pattern defined by replicated code realising the same concern. In other words, the given concern is implemented by similar pieces of code in different components. These duplications can occur, for instance, as a result of copy and paste practices and they increase the overall costs of maintenance activities [63]. Copy Cat also occurs when pieces of code implementing such concern are almost identical, varying only in small details. In either similar or identical code, every time one piece of concern code is modified, other copies are likely to require similar modifications.

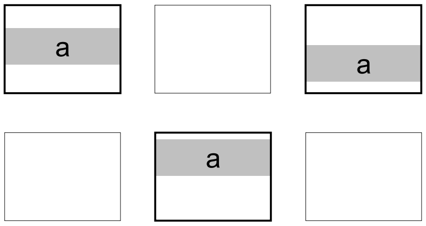
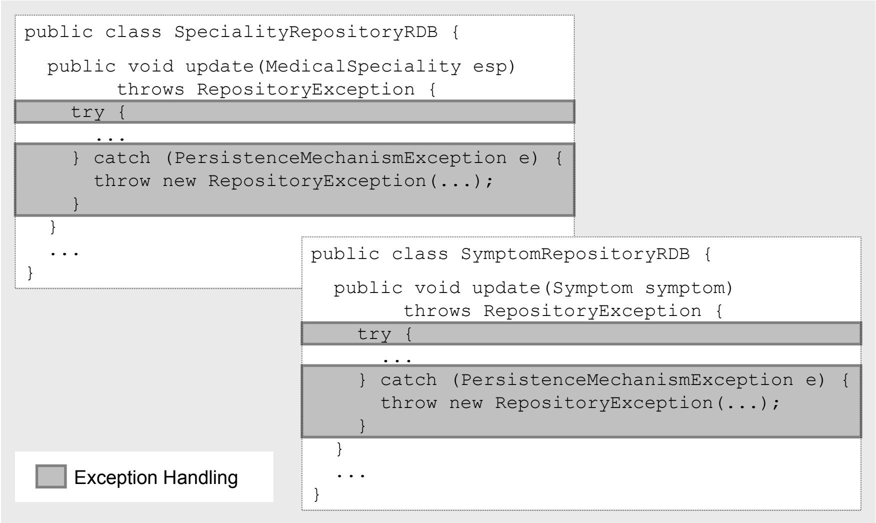


Figure 4-15: Abstract representation of Copy Cat

The abstract representation of Copy Cat is presented in Figure 4-15. The label 'a' in different boxes of this figure indicates that the shadowed areas contain similar code. To formalise this crosscutting pattern, we define the new relation *similar(con, m1, m2)*. This function verifies whether two members, *m1* and *m2,* of two different components have similar code assigned to the concern *con*. As presented in Section 3.1, the set of members realising a concern *con* is defined by *M(con)* and the set of members of a component *c* is defined by *M(c)*. Then, Copy Cat is expressed as:

$$\text{CopyCat(con)} \Leftrightarrow \exists\ c\ \exists\ d \in C(\text{con})\ \exists\ m \in (M(c) \cap M(\text{con}))$$
$$\exists\ n \in (M(d) \cap M(\text{con}))\ \text{similar(con, m, n)} \wedge c \neq d$$

To illustrate an instance of this crosscutting pattern, we rely on the partial source code of two classes of the Health Watcher system (Figure 4-16). The code of these classes implementing the Exception Handling concern is highlighted in Figure 4-16. In a closer look at this source code, we verified that these two pieces of exception handling code are exactly the same. In fact, there are 29 pieces of code like these two scattered over six of the Health Watcher classes. For this reason, Exception Handling is classified as Copy Cat in this system.



```
public class SpecialityRepositoryRDB {
  public void update(MedicalSpeciality esp)
        throws RepositoryException {
    try {
      ...
    } catch (PersistenceMechanismException e) {
      throw new RepositoryException(...);
    }
  }
  ...
}
```

```
public class SymptomRepositoryRDB {
  public void update(Symptom symptom)
        throws RepositoryException {
    try {
      ...
    } catch (PersistenceMechanismException e) {
      throw new RepositoryException(...);
    }
  }
  ...
}
```

Figure 4-16: Exception Handling as Copy Cat

4.5.2 Dolly Sheep

We identified a special type of the Copy Cat crosscutting pattern (Section 4.5.1) when the concern is also classified as Black Sheep (Section 4.2.1). In this particular case, the crosscutting pattern is called *Dolly Sheep* since it represents a “replicated sheep”. In fact, Dolly Sheep is not a completely new crosscutting pattern since it is defined by the composition of two others. This specific pattern composition is interesting for us because it recurrently appears in the analysed systems (Section 6.3). Moreover, it illustrates how a new crosscutting pattern can be defined based on the composition of two others. Figure 4-17 shows the abstract structure of Dolly Sheep. Similarly to Copy Cat, shadowed areas labelled ‘a’ indicate similar pieces of concern code. Moreover, we can observe that only small parts of the components are used to realise the analysed concern (i.e., the Black Sheep key property).

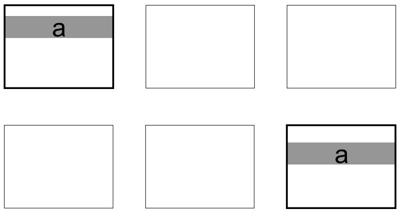


Figure 4-17: Abstract representation of Dolly Sheep

The formalisation of Dolly Sheep is straightforwardly defined (below). It states that a concern is Dolly Sheep if, and only if, this concern is both Black Sheep and Copy Cat. That is, their respective non-empty sets of components have exactly the same elements. It is important to note that, for every crosscutting pattern its respective set includes all components in *C(con)* as defined earlier in Section 4.2. The formal definition of Black Sheep and Copy Cat are presented in Sections 4.2.1 and 4.5.1, respectively. The Exception Handling concern presented in Figure 4-16 is an instance of Dolly Sheep, in addition to Copy Cat. In that figure, only small parts of some methods are used to realise Exception Handling.

$$\text{DollySheep(con)} \Leftrightarrow \text{BlackSheep(con)} = \text{CopyCat(con)} \neq \varnothing$$

4.5.3 Data Concern

The third crosscutting pattern of this section, *Data Concern* (or *Brain*), represents a crosscutting concern mainly composed of data. Data may be represented, for instance, by class variables and fields in an object-oriented design. Data Concern has no (or very little) associated behaviour. Behaviour can be associated with methods and constructors in an object-oriented design. In fact, existing behaviour in a Data Concern instance is due to the existence of data accessors (e.g., get and set methods).

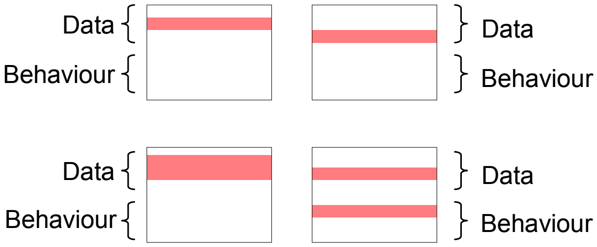


Figure 4-18: Abstract representation of Data Concern

Figure 4-18 shows the abstract representation of this crosscutting pattern. Note that, there is a logical division between data and behavioural elements of each box (i.e., component). Moreover, one component has a *small* piece of behaviour assigned to the analysed concern, which is accepted in the definition of this crosscutting pattern. However, most of the shadowed areas in Figure 4-18 represent data.

The formal definition of Data Concern uses two additional expressions: *data(m)* and *dataAccessor(m)*. These relations return Boolean values indicating whether the member $m \in M(c)$, is data (e.g., field or class variable) or a data accessor, respectively. A data accessor is an operation with the sole purpose of either getting the value from or setting a value into an attribute. Given these definitions, a concern is classified as Data Concern if, and only if, all members realising the given concern are data or data accessors.

$$\begin{aligned}\text{DataConcern(con)} \Leftrightarrow \forall\, c \in \text{C(con)}\ \forall\, m \in (\text{M(c)} \cap \text{M(con)}) \\ \text{data(m)} \vee \text{dataAccessor(m)}\end{aligned}$$

4.5.4 Behavioural Concern

The last crosscutting pattern defined in this chapter, *Behavioural Concern* (or *Brainless*), identifies a concern only composed of behaviour (e.g., methods and constructors in an object-oriented design). Behavioural Concern has no associated data as illustrated by its abstract representation in Figure 4-19.

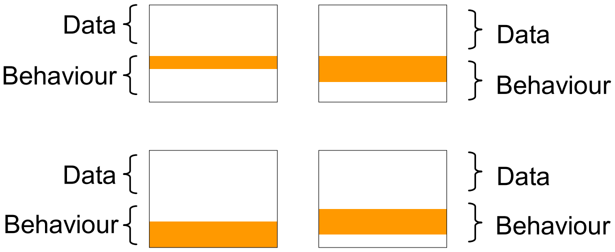


Figure 4-19: Abstract representation of Behavioural Concern

As a concrete instance, Figure 4-20 presents an implementation of the Prototype design pattern [64]. This instance is part of a library of design pattern implementations developed by Hannemann and Kiczales [80]. As we observe in the design of Figure 4-20, only methods (i.e., different implementations of `clone()`) are used to realise Prototype. Regarding the formal definition of Behavioural Concern, we rely on the previously defined *data(m)* expression (Section 4.5.3). Then, a concern is classified as

Behavioural Concern (below) when there is no member *m* realising this concern in any component, so that *m* is data.

$$\text{BehaviouralConcern(con)} \Leftrightarrow \forall\ c \in C(\text{con})\ \nexists\ m \in (M(c) \cap M(\text{con}))\ \text{data}(m)$$

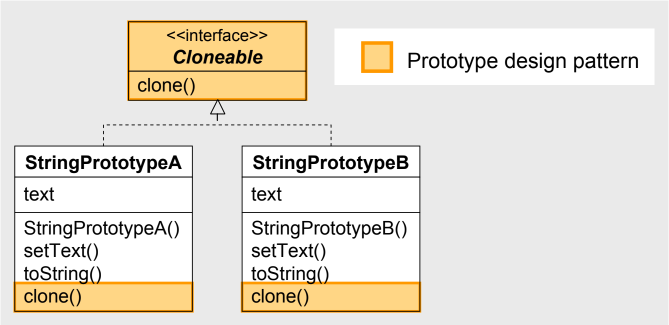


Figure 4-20: The Prototype design pattern as Behavioural Concern

4.6 Correlating Crosscutting Patterns

The previous sections of this chapter presented thirteen crosscutting patterns grouped into four categories. This section elaborates on the relationships among the definitions of different patterns. In other words, the crosscutting patterns were individually presented in different sections and correlations among their definitions were barely discussed so far. This section addresses this issue by discussing *orthogonal* and *overlapping* definitions of the crosscutting patterns. We call *orthogonal patterns* the case where two crosscutting patterns cannot simultaneously classify the same concern. On the other hand, the presence of a particular crosscutting pattern may also imply that other patterns are found in the same case, i.e., these patterns have *overlapping* definitions. The goal of documenting these orthogonal and overlapping definitions is to help understanding the whole picture of crosscutting patterns. This global view supports, for instance, the analysis of more complex concerns which entail more than one crosscutting pattern in their structure.

Inspired by the relationships among design patterns presented in Gamma's book [64], Figure 4-21 shows orthogonal and overlapping definitions of crosscutting patterns. Orthogonal definitions are represented by a 'cannot be' arrow connecting two patterns. For instance, a concern classified as Black Sheep (Section 4.2.1) or Dolly Sheep (Section 4.5.2) cannot be classified as Octopus (Section 4.2.2). This property is reasonable since a concern that affects only small parts of the system components (definition of Black Sheep and Dolly Sheep) cannot be partially well-modularised in a component (definition of the Octopus' body). The definition of the Octopus' body also hinders the simultaneous existence of Octopus with Behavioural Concern and Data Concern. The explanation is also due to the need of most operations and attributes of a class are realising the Octopus' body. Hence, the Octopus definition imposes operations and attributes to at least one class composing the given concern – which conflicts with the definitions of both Behavioural Concern and Data Concern. Analogous reasoning is applied to God Concern with respect to Black Sheep, Dolly Sheep, Behavioural Concern, and Data Concern.

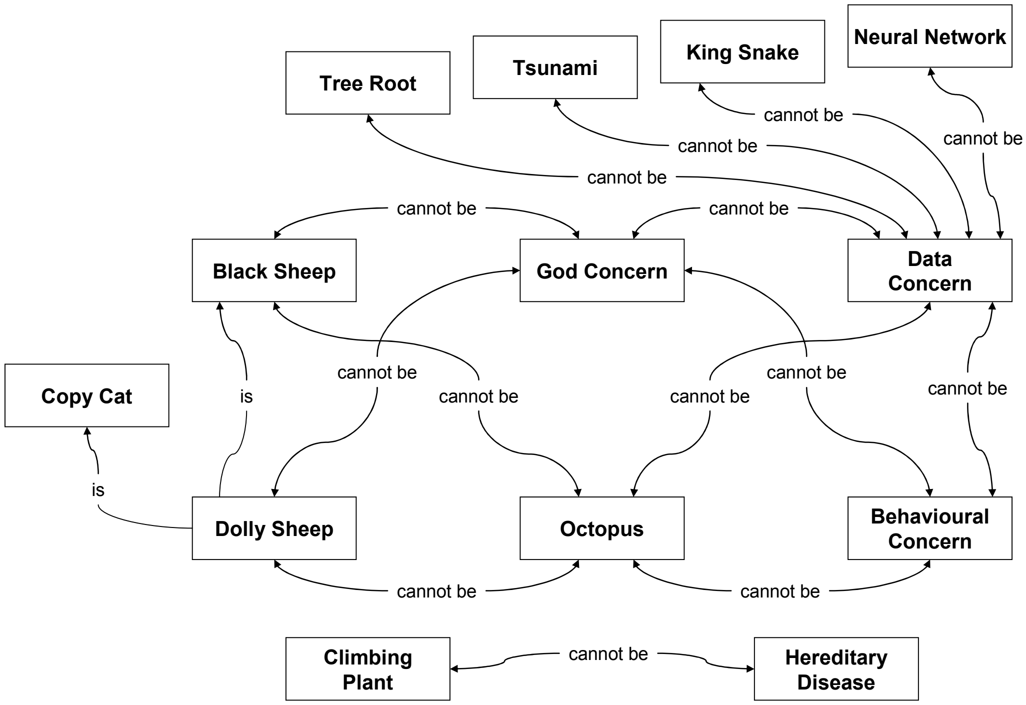


Figure 4-21: Crosscutting pattern relationships.

The definition of the Data Concern crosscutting pattern is particularly orthogonal to many other patterns' definitions. For obvious reason, a concern classified as Data Concern cannot be Behavioural Concern. Moreover, Data Concern has an orthogonal definition with respect to all crosscutting patterns in the category of Communicative Concerns (Section 4.4). This constraint is a result of the lack of operations in Data Concern. Operations are the key elements for the existence of high coupling among components. For instance, without operations, it is unlikely that a component realising the concern would connect to many other components as required by the Tsunami definition. Similarly, the chain of connected components, required by the King Snake definition, is also unlikely without an operation calling another, which in turn, should call another operation.

In addition to conflicts, Figure 4-21 also makes explicit the overlapping in the definitions of three crosscutting patterns: Copy Cat, Black Sheep, and Dolly Sheep. In fact, the relationship among these crosscutting patterns is due to the way Dolly Sheep is defined, i.e., a concern is Dolly Sheep if, and only if, it is both Copy Cat and Black Sheep. It should be noticed that the direction of the relationships is important in this case. In other words, a concern classified as Dolly Sheep is also Copy Cat and Black Sheep. However, an instance of Copy Cat or Black Sheep is not necessarily classified as Dolly Sheep.

The lack of relationships (i.e., arrows) connecting two crosscutting patterns means that there is no orthogonal or overlapping dependency in their definitions. For instance, all Communicative Concerns do not impose restrictions in their definitions with respect to other crosscutting patterns apart from Data Concern (discussed before). Hence, a concern possessing the structure of a crosscutting pattern in this category may also be classified according to other crosscutting patterns, such as Octopus and Climbing Plant. Although no constraint is imposed by some patterns' definitions, instances of specific crosscutting patterns might be commonly observed together.

4.7 Summary

In this chapter, we have presented a preliminary catalogue of thirteen patterns of crosscutting concerns identified by performing concern analysis of several heterogeneous systems (Section 4.1). The crosscutting patterns were classified in four categories (Sections 4.2 to 4.5) according to their common characteristics. These crosscutting patterns were inspired by Ducasse, Girba, and Kuhn [41] who identified four patterns of concerns – *Well-Encapsulated*, *Crosscutting*, *Black Sheep*, and *Octopus* – based on a generic technique, called Distribution Map. Our work complements their work by formalising Octopus and Black Sheep and by refining crosscutting into eleven new crosscutting patterns. We have not claimed that this set of crosscutting patterns is complete. In fact, we discuss the definition of further crosscutting patterns in Section 9.3.

This chapter also presented orthogonal and overlapping relationships observed in the given definitions of the crosscutting patterns (Section 4.6). To investigate the incidence of crosscutting patterns in architecture and design models, we perform in Chapters 7 and 8 empirical studies relying on three different applications. Moreover, typical combinations of the crosscutting patterns are further discussed in these evaluation chapters with respect to their positive and negative impact on architecture and design stability.

The next chapter (Chapter 5) discusses heuristic strategies to detect crosscutting patterns in software systems. These strategies combine information from a complementary set of concern-based analysis techniques, including traditional [29, 51, 95] and concern metrics [46, 57, 134], concern-based clone detection [18], and concern-based dependency between components [57]. Chapter 5 also presents a tool (Section 5.6) which, relying on the heuristic detection technique, automates the identification of crosscutting concerns and the documented crosscutting patterns.

5. Heuristic Detection of Crosscutting Patterns

As discussed in the background chapter, metrics and heuristic analysis are traditionally the main mechanisms for assessing design quality attributes, such as modularity and stability [27, 29, 95, 111]. However, there is not much knowledge on heuristic analysis for assessing design stability based on concern properties, even though it is widely recognised that the identification and modularisation of crosscutting concerns are important software design activities [88, 123, 129]. More recently, there is a growing body of relevant work in the software engineering literature focusing either on concern representation techniques [41, 52, 129] or on concern measurement tools [58, 129]. A number of concern metrics have also been recently proposed (see Section 2.2). All these tools and techniques are motivated by empirical studies which confirm that crosscutting concerns can lead to design flaws and can hinder design stability [44, 56, 73]. However, there is still a lack of higher-level heuristic techniques, such as *detection strategies* [95, 111], to support concern-sensitive analysis.

Based on evidence that not all types of crosscutting concerns are harmful [44, 52], Chapter 4 described thirteen patterns of crosscutting concerns (or crosscutting patterns). These patterns capture characteristics of crosscutting concerns that have been found to widely occur in a number of different systems. This chapter presents a heuristic detection technique to support the process of identifying such crosscutting patterns. This technique is supported by a prototype tool called ConcernMorph (Section 5.6). The proposed heuristic technique combines detection strategies and complementary algorithms to detect all categories of crosscutting patterns presented in Chapter 4. The identification of crosscutting patterns requires a first step which is the recognition of crosscutting concerns. Therefore, we also present detection strategies (Section 5.1) that combine traditional and concern metrics aiming at identifying crosscutting concerns.

After crosscutting concerns have been identified, we propose metrics-based detection strategies to identify which crosscutting concerns match the category of flat crosscutting patterns (Section 5.2). Moreover, tree-based algorithms and graph-based algorithms complement the detection strategies by supporting the identification of crosscutting concerns in the categories of inheritance-wise concerns (Section 5.3) and communicative concerns (Section 5.4), respectively. Finally, we also propose concern-based clone detection [18] which, combined with the previous heuristic techniques, can be used to identify the other crosscutting patterns (Section 5.5). Section 5.6 presents the tool that supports the proposed heuristic technique and Section 5.7 concludes this chapter.

5.1 Identifying Crosscutting Concerns

Inspired by the detection strategies proposed by Marinescu [111, 112], we define specific strategies to characterise crosscutting concerns. Marinescu's work composes traditional module-driven metrics [29, 95] into rules aiming to heuristically detect a set of traditional design flaws, such as bad smells [63]. In this section, the original detection strategy mechanism discussed in Section 2.3 is extended with recently proposed concern metrics [44, 57]. We call these extended detection rules *concern-sensitive detection strategies*. Section 5.1.1 presents the overall structure of a concern-sensitive detection strategy. Then, Section 5.1.2 illustrates a set of detection strategies with the goal of identifying crosscutting concerns. The identification of crosscutting concerns is required prior to their classification in crosscutting patterns. Additional detection strategies are also defined later in this chapter (Sections 5.2 and 5.5).

5.1.1 Structure of Concern-Sensitive Detection Strategies

This section presents the structure of concern-sensitive detection strategies. These mechanisms support heuristic analysis of design modularity and stability focusing on the system concerns. Each proposed detection strategy is heuristically defined in terms of combined information collected from concern metrics and traditional modularity metrics.

Table 5-1 presents three traditional metrics which are used by the proposed detection strategies. The concern metrics used in this chapter and already presented in Section 3.3 are summarised in Table 5-2. Both Tables 5-1 and 5-2 include the metric names (1st column) and their short descriptions (2nd column). A concern-sensitive detection strategy is defined by a heuristic expression which embodies knowledge about the concern realisation in a specific design.

Table 5-1: Traditional software engineering metrics [134]

Metric	Definition
Number of Components (NC) [134]	Counts the number of components of the system.
Number of Attributes (NOA) [134]	Counts the number of attributes of each component.
Number of Operations (NOO) [134]	Counts the number of operations of each component.

Table 5-2: Concern metrics

Metric	Definition
Lack of Concern-based Cohesion (LCC)	Counts the number of concerns realised by the target component.
Concern Diffusion over Components (CDC)	Counts the number of components related to a concern.
Number of Concern Attributes (NOCA)	Counts the number of attributes that realise a concern.
Concern Diffusion over Operations (CDO)	Counts the number of operations that realise a concern.

Detection strategies are expressed using conditional statements in the form: *IF <condition> THEN <consequence>*. The condition part encompasses one or more metrics' outcomes related to the design concern under analysis. If the condition is not satisfied, then the concern analysis is concluded and the concern classification is not refined. In case the condition holds, the consequence part plays the role of describing a change or refinement of the target concern classification. In some situations, the result of detection strategies can be directly interpreted as a warning of design flaw. In this case, this warning also provides information that helps the designers to concentrate on certain concerns or parts of the design which are potentially problematic. The following subsection presents detection strategies for identifying crosscutting concerns. More elaborated detection strategies are presented later for classifying crosscutting concerns in some categories of crosscutting patterns (Sections 5.2 and 5.5).

5.1.2 Basic Rules for Crosscutting Concern Detection

Detection strategies presented in this section identify the basic ways a concern can manifest itself through the software modularity units. Our goal is to identify crosscutting concerns before classifying them according to the proposed crosscutting patterns. Hence, the detection strategies of this section are structured in such a way that a concern is systematically classified into a more specialised category, representing a more severe bad symptom, until it is found to be crosscutting. A concern can be classified into one (or more) of six basic concern categories (Figure 5-1): Isolated, Tangled, Little Scattered, Highly Scattered, Well Encapsulated, and Crosscutting. We do not consider these classifications crosscutting patterns because they represent the basic characteristics of a concern which is not necessarily crosscutting.

A *tangled concern* is interleaved with other concerns in at least one component (i.e., class or aspect). If the concern is not tangled in any component, it is considered *isolated*. A *scattered concern* spreads over multiple components. Our classification makes a distinction between *highly scattered* and *little scattered* concerns based on the number of affected components. A concern is *crosscutting* only if it is both tangled with other concerns and scattered over multiple system components. As we demonstrate later, even little scattered concerns might be considered crosscutting in some situations. If a concern is not crosscutting, it is said to be *well-encapsulated*. Crosscutting concerns generate warnings of inadequate separation of concerns and, consequently, opportunities for the detection of crosscutting patterns.

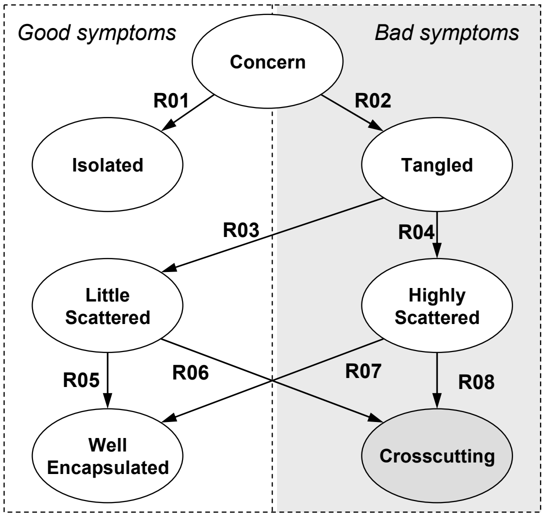


Figure 5-1: Basic classification of concerns

Figure 5-1 presents the transitions between the basic concern classifications using labelled arrows (R01, R02…). Each arrow represents a concern-sensitive detection strategy. This diagram makes explicit the application order for the detection strategies. This order was defined based on our empirical knowledge on the use of concern metrics. For instance, we usually check whether a concern is tangled or not (by means of CDLOC) before proceeding with any further concern analysis. Once a concern is found to be tangled, we verify how scattered this concern is (CDC supports this analysis). Finally, we go for a more detailed analysis (e.g., using CDO and NOCA) in order to make sure that this concern is indeed crosscutting.

Figure 5-2 shows concern-sensitive detection strategies (also called *rules*, for short) corresponding to the diagram of concern classification presented in Figure 5-1. The first two rules, R01 and R02, use the metric Lack of Concern-based Cohesion (LCC) to classify the concern as isolated or tangled. LCC counts the number of concerns per component (Table 5-2). If the LCC value is one for every component realising the analysed concern, it means that there is only the analysed concern in these components and, therefore, the concern is isolated. However, if LCC is higher than one in at least one component, it means that the concern is tangled with other concerns in that component.

R01 - *Isolated*:
if LCC = 1 **for every component with** Concern **then** Concern is Isolated

R02 - *Tangled*:
if LCC > 1 **for at least one component then** Concern is Tangled

R03 - *Little Scattered*:
if CDC / NC of Concern < 0.5 **then** Tangled Concern is Little Scattered

R04 - *Highly Scattered*:
if CDC / NC of Concern ≥ 0.5 **then** Tangled Concern is Highly Scattered

R05 - *Well Encapsulated*:
if (NOCA / NOA ≥ 0.5) **and** (CDO / NOO ≥ 0.5) **for every component**
then Little Scattered Concern is Well-Encapsulated

R06 - *Crosscutting*:
if (NOCA / NOA < 0.5) **and** (CDO / NOO < 0.5) **for at least one component**
then Little Scattered Concern is Crosscutting

R07 - *Well Encapsulated*:
if (NOCA / NOA ≥ 0.5) **and** (CDO / NOO ≥ 0.5) **for every component**
then Highly Scattered Concern is Well-Encapsulated

R08 - *Crosscutting*:
if (NOCA / NOA < 0.5) **and** (CDO / NOO < 0.5) **for at least one component**
then Highly Scattered Concern is Crosscutting

Figure 5-2: Detection strategies for crosscutting concerns

Rules R03 and R04 (Figure 5-2) verify whether a concern, besides being tangled, is scattered over multiple components. These concern-sensitive detection strategies use the metrics Concern Diffusion over Components (CDC) and Number of Components (NC) in order to calculate the percentage of system components affected by the concern of interest. Based on this percentage, the concern is classified as highly scattered or little scattered. Developers should be aware of highly scattered concerns because they can potentially cause design flaws, such as Shotgun Surgery [63] (Section 2.5.2). Note that the selection of threshold values is one of the most sensitive parts in a detection strategy. We use some default threshold values (e.g., 50% in R03 to R08) for detection strategies in this section. However, our tool (Section 5.6) allows developers to set customised values for each concern-sensitive detection strategy.

As the evidence of tangling and scattering might mean different levels of crosscutting, both highly scattered and little scattered concerns can be classified either as well-encapsulated or as crosscutting concerns. Rules R05 and R06 decide whether a little scattered concern is a well-encapsulated or a crosscutting concern. Rules R07 and R08 perform similar analysis for a highly scattered concern. These four rules rely on two concern metrics presented in Table 5-2, namely Number of Concern Attributes (NOCA) and Concern Diffusion over Operations (CDO). They also use two size metrics (Table 5-1): Number of Attributes (NOA) and Number of Operations (NOO). The combination of these metrics calculates, for each component, the percentage of attributes and operations which realises the concern being analysed.

The role of the detection strategies R05 and R07 is to detect components that dedicate a large number of attributes and operations (more than 50%) to realise the dominant concern. If a concern is dominant in all components where it is, the concern is classified as well-encapsulated. The reasoning behind this classification is that a concern is not harmful to components in which it is dominant, and, therefore, it does not need to be removed. Analogously, a concern is classified as crosscutting (rules R06 and R08) if the percentage of attributes and the percentage of operations related to the concern are low (less than 50%) in at least one component. In this case, the concern is classified as crosscutting since it is located in at least one component which has another concern as dominant.

5.2 Detection Strategies for Flat Crosscutting Patterns

Chapter 4 defined thirteen patterns of crosscutting concerns organised in four categories. This section illustrates how concern-sensitive detection strategies can be used to identify concerns matching patterns in the first category: Flat Crosscutting Patterns (Section 4.2). This category includes three crosscutting patterns, namely Black Sheep, Octopus, and God Concern, briefly described below. Unlike the formal definitions of crosscutting patterns presented in Chapter 4, the detection strategies represent heuristic and operational ways of identifying theses patterns. Note that the crosscutting concerns must be first identified (Section 5.1) before applying the strategies presented in this section.

> *Black Sheep is a concern that crosscuts the system, but is realised by very few elements in different components.*
>
> *Octopus is a concern which is well encapsulated in a few components, but also spreads across other components.*
>
> *God Concern is a concern which encompasses too much scattered functionality of the system.*

From these definitions, we verify that Black Sheep, Octopus, and God Concern are actually specialised categories of crosscutting concerns. For this reason, a concern must be previously classified as crosscutting using the rules presented in Section 5.1. Then, these concerns can be further investigated regarding their crosscutting pattern. Figure 5-3 shows three detection strategies – R09, R10, and R11 – which aim at identifying Black Sheep, Octopus, and God Concern, respectively. Figure 5-3 also defines two conditions A (*Little Dedication*) and B (*High Dedication*) used in these rules. We explicitly separate the conditions from the detection strategies not only to make the rules easier to understand but also to reuse the conditions in more than one detection strategy. By applying these rules, a concern previously classified as crosscutting (Section 5.1) is thoroughly inspected in terms of how it manifests its crosscutting nature over the system's components.

The detection strategy R09 classifies a crosscutting concern as Black Sheep if all components which have this concern dedicate only a few percentage points of attributes and operations (the default value is less than 33%) to the concern realisation. The percentage of attributes and operations is calculated based on two concern metrics – Number of Concern Attributes (NOCA) and Concern Diffusion over Operations (CDO) – and two size metrics: Number of Attributes (NOA) and Number of Operations (NOO). We choose the default value of 33% for the little dedication condition based on (i) threshold values used by other authors [41, 95], and (ii) a meaningful ratio that represents the definition of Black Sheep: '*realised by very few elements*'. Moreover, Lanza and Marinescu [95] suggest the use of meaningful threshold values, such as 0.33 (1/3), 0.5 (1/2), and 0.67 (2/3). Among these values, 1/3 seems the most appropriate one for the Black Sheep definition.

```
Condition A - Little Dedication: (NOCA / NOA < 0.33) and (CDO / NOO < 0.33)
Condition B - High Dedication: (NOCA / NOA ≥ 0.67) and (CDO / NOO ≥ 0.67))
R09 - Black Sheep:
if (Little Dedication) for every component with Concern
then Crosscutting Concern is Black Sheep
R10 - Octopus:
if ) ( (Little Dedication) for at least 1 component with Concern )
   and ( (High Dedication) for at least 1 component with Concern )
then Crosscutting Concern is Octopus
R11 – God Concern:
if ((CDC / NC) > 0.33) and (High Dedication) for most components with Concern
then Crosscutting Concern is God Concern
```

Figure 5-3: Detection Strategies for Black Sheep, Octopus, and God Concern

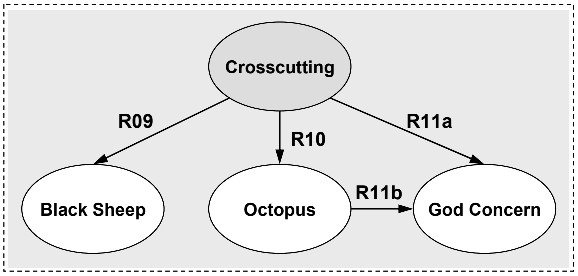


Figure 5-4: Classification of Flat Crosscutting Patterns

The next rule R10 verifies if a crosscutting concern not classified as Black Sheep is a potential Octopus. Figure 5-4 makes explicit that a concern classified as Black Sheep cannot be Octopus. Moreover, this figure indicates that an Octopus concern can later be classified as God Concern. According to the detection strategy R10, a concern is classified as Octopus if at least one component has to be little dedicated (tentacles of the octopus) and another has to be highly dedicated (body of the octopus) to the concern realisation. We define a component as highly dedicated to the concern realisation (Condition B) when the percentage of attributes and operations is, for instance, higher than 67%. Again, default threshold values try to be a meaningful approximation of the Octopus' definition.

The third detection strategy (R11) aims at finding God Concern. A crosscutting concern is classified as God Concern if two conditions are satisfied: (i) the given concern is realised by more than 33% of the system's components and (ii) most of these components are highly dedicated to the concern realisation (Condition B). To verify the first condition, this rule calculates the percentage of components realising the given concern (i.e., CDC / NC). With respect to the second condition, not all components in the system are highly dedicated to the concern realisation, since this concern was previously classified as crosscutting (Section 5.1). In other words, this concern needs to be previously classified as crosscutting in at least one component (see rules R06 and R08 in Section 5.1.2). Therefore, we just need to verify if the concern is dominant in more than 50% of the components where it is located.

We would like to emphasise that a detection strategy presented in this chapter may not be 100% in compliance with the corresponding pattern's formal definition presented in Chapter 4. That is, the detection strategies (and algorithms to be presented in the next sections) are a heuristic way we used to approximate the intention given by the formal definition of a crosscutting pattern. Additional detection means could be further defined, for instance, using different strategies.

5.3 Tree-based Algorithms to Detect Inheritance-wise Concerns

This section presents heuristic strategies to detect crosscutting patterns in the category of Inheritance-wise Concerns. The two crosscutting patterns of this category, Climbing Plant and Hereditary Disease, were presented in Section 4.3. Their definitions can be summarised as follows.

> *Climbing Plant is a crosscutting concern which is realised by all components in an inheritance tree.*
>
> *Hereditary Disease is a crosscutting concern which is realised by some components in an inheritance tree.*

Unlike the previously discussed Flat Crosscutting Patterns, the identification of Inheritance-wise Concerns does not rely on concern-sensitive detection strategies, i.e., rules composed of concern metrics. Instead, both Climbing Plant and Hereditary Disease crosscutting patterns can be identified by means of algorithms for searching in trees. Using pseudo code, Figures 5-5 and 5-6 show algorithms to heuristically detect Climbing Plant and Hereditary Disease, respectively.

The algorithm for detecting Climbing Plant presented in Figure 5-5 tries to find an inheritance tree with all components realising the given concern. It basically follows two steps. First, it identifies components that realise the given concern passed as a parameter (lines 1 to 3). These components also need to be specialised by sub-components (line 4). This last requirement ensures that they are roots of some inheritance trees. Second, every sub-component *c'* of a root component *c* is inspected to make sure it realises the given concern (lines 5 to 8). If all sub-components of a root component present this characteristic, the given concern holds for the Climbing Plant crosscutting pattern. On the other hand, if any tree of components could be found with these particular characteristics, the concern is not classified as Climbing Plant (line 11).

```
01 boolean detectClimbingPlant(Concern, Components) {
02   for all (component c in Components) do {
03     if (component c realises Concern) and
04         (component c is specialised by sub-components) then {
05       if (every component c′ which specialises c
06             also realises Concern) then {
07           return true;
08       }
09     }
10   }
11   return false;
12 }
```

Figure 5-5: Algorithm for Climbing Plant detection

The Hereditary Disease algorithm presented in Figure 5-6 tries to find an inheritance tree which has at least one component not realising the concern. However, the root of the inheritance tree must realise the concern under analysis. Similarly to Climbing Plant, the algorithm for Hereditary Disease looks for root components that realise the concern given as a parameter (lines 1 to 4). Then, it tries to find a disease-free node in the inheritance tree rooted at each root component. That is, a sub-component of the root component which does not realise the concern under analysis. If a disease-free node is found, the concern is classified as Hereditary Disease. Otherwise, the target concern does not hold for this crosscutting pattern.

```
boolean detectHereditaryDisease(Concern, Components) {
  for all (component c in Components) do {
    if (component c realises Concern) and
        (component c is specialised by sub-components) then {
      if (at least one component c′ which specialises c
          does not realise Concern) then {
        return true;
      }
    }
  }
  return false;
}
```

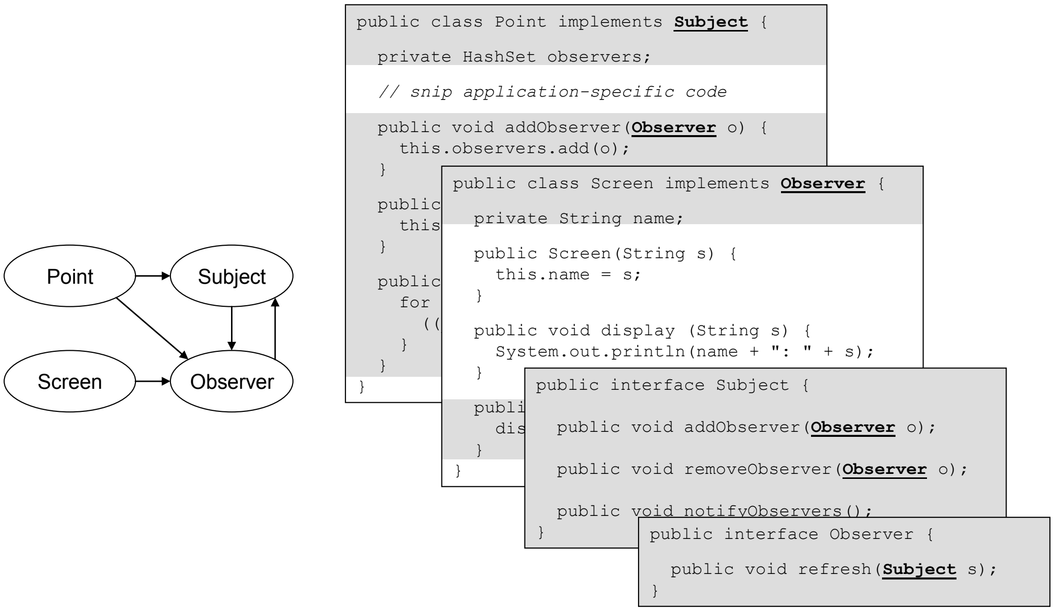
Figure 5-6: Algorithm for Hereditary Disease detection

5.4 Graph Search Algorithms to Detect Communicative Concerns

This section presents heuristic strategies for detecting the four Communicative Concerns: Tree Root, Tsunami, Neural Network, and King Snake. These crosscutting patterns briefly described below were presented in Section 4.4.

> *Tree Root is a crosscutting concern which creates many coupling connections converging to the trunk components.*
>
> *Tsunami is a crosscutting concern which creates many coupling connections diverging from the wave source components.*
>
> *Neural Network is a crosscutting concern which creates a net of inter-connected components.*
>
> *King Snake is a crosscutting concern which creates a chain of connected components.*

To identify crosscutting patterns in the category of Communicative Concerns, the first step is to build a graph of connections, also called *graph of concern dependencies*, for each concern under analysis. The vertices of this graph are all components realising the analysed concern. Each edge represents a coupling connection between a pair of components (both components realising the concern). Figure 5-7 presents the graph of concern dependencies (on the left-hand side) for an implementation of the Observer design pattern [64], partially presented on the right-hand side. This implementation was developed by Hannemann and Kiczales [80]. Coupling connections are only taken into consideration if they are located in code fragments assigned to the concern (shadowed code in Figure 5-7). For instance, the shadowed code of the Point class refers to the Subject and Observer interfaces. Hence, the corresponding Point node in the graph has two edges connecting it to the nodes of these two interfaces.



```
public class Point implements Subject {

  private HashSet observers;

  // snip application-specific code

  public void addObserver(Observer o) {
    this.observers.add(o);
  }

  public
    this
  }

  public
    for
      ((
    }
  }
}
```

```
public class Screen implements Observer {

  private String name;

  public Screen(String s) {
    this.name = s;
  }

  public void display (String s) {
    System.out.println(name + ": " + s);
  }

  publi
    dis
  }
}
```

```
public interface Subject {

  public void addObserver(Observer o);

  public void removeObserver(Observer o);

  public void notifyObservers();
}
```

```
public interface Observer {

  public void refresh(Subject s);
}
```

Figure 5-7: Graph of concern dependencies for the Observer design pattern

After the creation of a graph of concern dependencies for each concern, our heuristic strategy to identify crosscutting patterns uses variations of two graph search algorithms: Breadth First Search (BFS) and Depth First Search (DFS) [91]. Tree Root and Tsunami can be found by means of an adapted BFS algorithm presented in Figure 5-8. Our BFS variant records the number of vertices that can be reached by each vertex (lines 11 and 15). Apart from this, the algorithm follows the same basic steps of an ordinary Breadth First Search algorithm.

Figure 5-9 illustrates a simulation of this BFS algorithm in the sample graph of Figure 5-7. Vertices reached by a given vertex have their values placed between brackets when they become grey and, then, updated when they become black (visited). The algorithm is recurrently called until all vertices become black. The recorded values indicate the number of waves for a particular wave source when this algorithm is used to detect Tsunami. Moreover, the wave source of Tsunami is represented by vertices with the highest assigned value after the BFS execution has finished. The Tree Root crosscutting pattern can be identified using exactly the same algorithm. However, for Tree Root detection, the input graph must have inverted edges. Then, in the Tree Root case, vertices with the highest value represent the trunk. The value of each vertex indicates its number of feeders.

```
void bfs(Graph) {
 for all (vertex v of Graph) {
  v is first painted white;
 }
 the graph root r is painted grey; its value is set 0;
 r is put in a queue Q;
 while (the queue Q is not empty) {
  a vertex u is removed from the queue;
  for all (white successor v of u) {
   v is painted grey; its value is set 0;
   the value of v is increased by one;
   v is added to the queue;
  }
  u is painted black;
  the u value is set one unit higher than
    the highest successor;
 }
}
```

Figure 5-8: Breadth First Search adapted for Tree Root and Tsunami detection

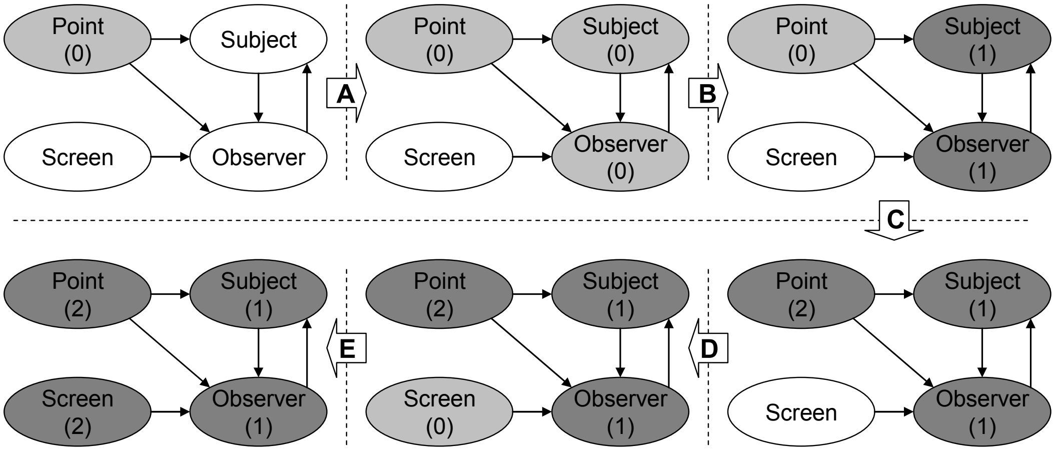


Figure 5-9: Breadth First Search for a graph of concern dependencies

The identification of the Neural Network and King Snake crosscutting patterns requires a directed acyclic graph (DAG), i.e., a directed graph with no directed cycles.

To achieve a DAG which matches our aim, we propose the algorithm presented in Figure 5-10 which is based on Depth First Search (DFS) [91]. This algorithm marks vertices it has already visited (lines 2 and 3 of Figure 5-10) and does not visit these vertices again. Therefore, it transforms the graph of concern dependencies into a DAG. Figure 5-11 simulates the execution of this DFS-based algorithm in our running example of a graph of concern dependencies. In this figure, only solid edges are included in the DAG (dotted edges may be included if they become solid).

```
01 void dfs-visit(Graph, Vertex) {
02   the Vertex is painted grey; its value is set 0;
03   for all (white successor v of Vertex) {
04     the value of v is increased by one;
05     dfs-visit(Graph, v);
06   }
07   the Vertex value is set one unit higher than
          the highest successor;
08   Vertex is painted black;
09 }

10 void dfs(Graph) {
11   all vertices of Graph are first painted white;
12   dfs-visit(Graph, root of Graph)
13 }
```

Figure 5-10: Depth First Search adapted for Neural Network and King Snake detection

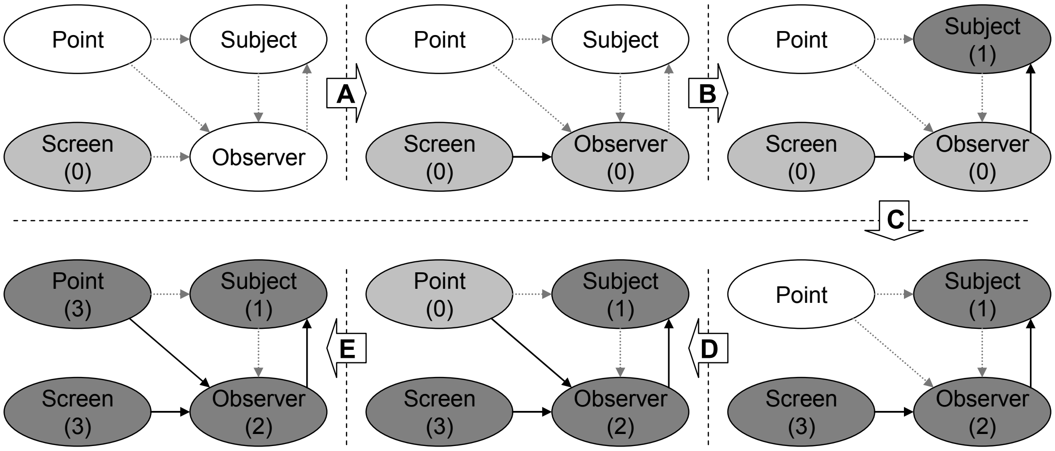


Figure 5-11: Depth First Search for a graph of concern dependencies

The resulting DAG (after Step E of Figure 5-11) includes all connected components and, so, it represents an instance of the Neural Network crosscutting pattern for the given concern. To identify the King Snake crosscutting pattern, we need a step further, though. As presented in Figure 5-11, our DFS variant marks vertices with the longest path (deepest) from another arbitrary vertex (lines 4 and 7 of Figure 5-10). Therefore, we retrieve components with the longest path to the deepest ones. These components represent the head of King Snake. The deepest components reached by each head component represent the King Snake tail. Taking the intermediary nodes, we also have the components composing the King Snake body.

5.5 Detecting Other Crosscutting Patterns

This section presents strategies towards the automated detection of the remaining four crosscutting patterns: Copy Cat, Dolly Sheep, Data Concern, and Behavioural Concern. These crosscutting patterns, summarised below, were presented and formalised in Section 4.5.

> *Copy Cat is a crosscutting concern which is realised by scattered replicated code.*
>
> *Dolly Sheep is a crosscutting concern which is realised by small pieces of scattered replicated code.*
>
> *Data Concern is a crosscutting concern which is mainly realised by attributes.*
>
> *Behavioural Concern is a crosscutting concern which is only realised by operations.*

In fact, we do not fully support the identification of Copy Cat and Dolly Sheep in our tool (Section 5.6). The reason for this is because we have not yet integrated our tool with techniques for code clone detection [7, 18, 42, 90, 106]. Moreover, this may not be worth to research since existing tools [7, 18, 106] already cope well with detection of replicated code. Copy Cat and Dolly Sheep require the use of clone detection techniques to find duplicated or similar pieces of concern code in different components.

Despite not being implemented, Figure 5-12 shows an algorithm using pseudo code aiming to detect the Copy Cat crosscutting pattern. This algorithm looks for two similar code fragments in different components (lines 2 to 5 in Figure 5-12).

Moreover, both code fragments must realise the concern under analysis (line 6). If these two conditions are satisfied, the concern is classified as Copy Cat (line 7). The same algorithm can be used to detect Dolly Sheep. However, in the Dolly Sheep case, this algorithm would only be called for concerns which were previously classified as Black Sheep (Section 5.2). This condition is required because, by definition, a Dolly Sheep concern has characteristics of both Copy Cat and Black Sheep (Section 4.5).

```
01 boolean detectCopyCat(Concern, Components) {
02   for all (pair of components c1, c2 in Components) do {
03     if (components c1 and c2 realise Concern) then {
04       if ( (code fragment f1 in c1) is similar to
05            (code fragment f2 in c2) ) and
06            (f1 and f2 realise Concern) then {
07         return true;
08       }
09     }
10   }
11   return false;
12 }
```

Figure 5-12: Algorithm for Copy Cat detection

The other two crosscutting patterns, Data Concern and Behavioural Concern, can be identified by means of concern-sensitive detection strategies (Section 5.1). Figure 5-13 presents two detection strategies, R12 and R13, to find these crosscutting patterns. The heuristic rule for Data Concern, R12, is based on two concern metrics (Table 5-2): Number of Concern Attributes (NOCA) and Concern Diffusion over Operations (CDO). It also uses two traditional size metrics (Table 5-1): Number of Attributes (NOA) and Number of Operations (NOO). Then, the heuristic strategy is, for every component, to investigate if many attributes and few operations are realising the concern. Note that a concern can be classified as Data Concern even if some operations are used to realise it. That is, the Data Concern definition also allows some operations, such as getting and setting methods, realising a concern of this pattern as described in Section 4.5. Moreover, the heuristic strategy may not be 100% accurate and, therefore, some operations which are neither getting nor setting methods may be misled by this strategy.

Figure 5-13 also shows a concern-sensitive detection strategy, R13, aiming to identify Behavioural Concern. A concern is classified by R13 in this crosscutting pattern if it is only composed of operations. For programs written in Java, for instance, Behavioural Concern indicates a concern which is only realised by methods and constructors. On the other hand, advice may also be considered as an operation in AspectJ systems. This decision, in fact, depends on the instantiation of specific language constructs using the framework meta-model presented in Chapter 3. To check whether a crosscutting concern is Behavioural Concern, the detection strategy R13 relies on two concern metrics: Number of Concern Attributes (NOCA) and Concern Diffusion over Operations (CDO). Then, R13 verifies if the NOCA value is zero and CDO is greater than zero for the analysed concern. If so, this crosscutting concern is classified as Behavioural Concern.

R12 – *Data Concern*:
if ((NOCA/NOA) > (CDO/NOO)) **for every component with** CONCERN
then CROSSCUTTING CONCERN is DATA CONCERN

R13 – *Behavioural Concern*:
if (NOCA = 0) **and** (CDO > 0) **for every component with** CONCERN
then CROSSCUTTING CONCERN is BEHAVIOURAL CONCERN

Figure 5-13: Detection Strategies for Data Concern and Behavioural Concern

5.6 ConcernMorph: a Tool for Crosscutting Pattern Detection

The detection of crosscutting patterns is a tedious task without proper tool support. This section describes ConcernMorph[5], a tool to automatic detect crosscutting patterns in Java programs. ConcernMorph is implemented as an Eclipse plug-in [47] and supports the concern-sensitive detection strategies and algorithms presented in the previous sections of this chapter. The tool also implements a set of concern metrics used by the detection strategies in the pattern detection. Section 5.6.1 presents the tool architecture and its main functionality. Section 5.6.2 shows some screenshots of the user interface.

5.6.1 The ConcernMorph Architecture

The detection of crosscutting patterns requires users to specify the projection of concerns into syntactic elements, such as classes, methods, and attributes. An existing tool, ConcernMapper [33], offers good support for this and is used by ConcernMorph. In other words, our tool relies on the projection of concerns to syntactic elements provided by ConcernMapper. Figure 5-14 shows the architecture of ConcernMorph

[5] Available for download at http://www.lancs.ac.uk/postgrad/figueire/concern/plugin/

and its relationships to ConcernMapper and the Eclipse Platform. Both tools, ConcernMorph and ConcernMapper, are coupled to the Eclipse Platform. ConcernMorph is also coupled to ConcernMapper. ConcernMorph defines two main modules (Figure 5-14): (i) *Metric Collector* which is responsible for computing concern metrics and (ii) *Rule Analyser* which applies heuristic rules to identify crosscutting patterns.

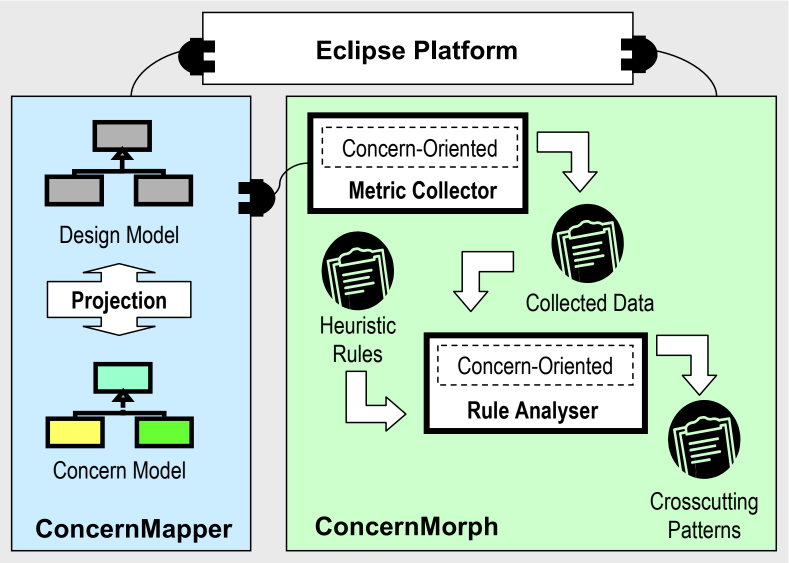


Figure 5-14: The ConcernMorph Architecture

The Metric Collector module of ConcernMorph implements six concern metrics and three traditional size metrics. Concern metrics (Chapter 3) quantify properties of concerns which may be realised by multiple components. Traditional metrics (Table 5-1), by contrast, measure properties of components. The supported concern and traditional metrics are:

- *Concern metrics*: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), Number of Concern Attributes (NOCA), Lack of Concern-based Cohesion (LCC), Number of Tangled Components (TC), and Number of Tangled Operations (TO). In particular, the last two concern metrics, TC and TO, have not been presented before. They count, respectively, the number of components and operations that a concern shares with other concerns. These concern metrics give extra information about the crosscutting concerns of the systems.

- *Traditional metrics*: Number of Components (NC), Number of Attributes (NOA), and Number of Operations (NOO).

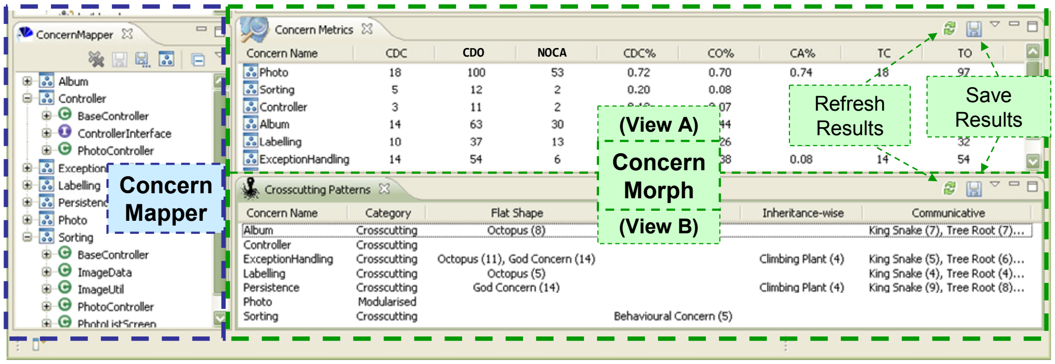
In the Rule Analyser module, each crosscutting pattern is defined as a heuristic rule. A rule combines a number of related metrics, thresholds, and algorithms to determine whether a case of crosscutting should be classified as an instance of the pattern. The current version of ConcernMorph supports twelve heuristic rules, but new rules can be easily included using the tool extension points. Based on the basic detection strategies (see Section 5.1), the first rule determines whether a concern is crosscutting. After the crosscutting concerns have been identified, the remaining rules classify them according to eleven crosscutting patterns in four categories (see Sections 5.2 to 5.5). The supported crosscutting patterns per category are:

- *Flat crosscutting patterns*: Black Sheep, Octopus, and God Concern
- *Inheritance-wise concerns*: Climbing Plant and Hereditary Disease
- *Communicative concerns*: King Snake, Tree Root, Tsunami, and Neural Network
- *Other crosscutting patterns*: Data Concern and Behavioural Concern

5.6.2 The ConcernMorph User Interface

Each module of ConcernMorph presented in Figure 5-14 adds a new view to the Eclipse Platform. Figure 5-15 presents the two views of ConcernMorph: Concern Metrics (View A) and Crosscutting Patterns (View B). The Concern Metrics view shows concern measurements while the Crosscutting Patterns view shows all instances of crosscutting patterns in the target system.

We use one of our case studies, called MobileMedia, in the illustrative example of Figure 5-15. In the Concern Metrics view of this figure, we see that the Photo concern, for instance, is scattered over 18 components (CDC), 100 operations (CDO), and 53 Attributes (NOCA). The Crosscutting Patterns view shows not only all instances of crosscutting patterns but also the number of components (between parentheses) taking part in each pattern instance. For example, the Album concern is classified as Octopus and 8 classes compose this Octopus instance (Figure 5-15).

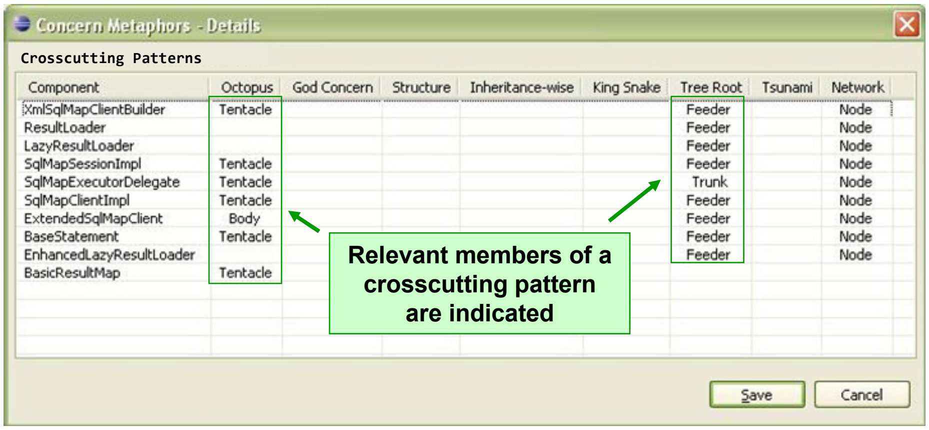


Concern Name	CDC	CDO	NOCA	CDC%	CO%	CA%	TC	TO
Photo	18	100	53	0.72	0.70	0.74	18	97
Sorting	5	12	2	0.20	0.08	[illegible]	[illegible]	[illegible]
Controller	3	11	2	[illegible]	0.07	[illegible]	[illegible]	[illegible]
Album	14	63	30	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Labelling	10	37	13	[illegible]	[illegible]	[illegible]	[illegible]	32
ExceptionHandling	14	54	6	[illegible]	[illegible]	0.08	14	54

Concern Name	Category	Flat Shape	[illegible]	Inheritance-wise	Communicative
Album	Crosscutting	Octopus (8)			King Snake (7), Tree Root (7)...
Controller	Crosscutting				
ExceptionHandling	Crosscutting	Octopus (11), God Concern (14)		Climbing Plant (4)	King Snake (5), Tree Root (6)...
Labelling	Crosscutting	Octopus (5)			King Snake (4), Tree Root (4)...
Persistence	Crosscutting	God Concern (14)		Climbing Plant (4)	King Snake (9), Tree Root (8)...
Photo	Modularised				
Sorting	Crosscutting		Behavioural Concern (5)		

Figure 5-15: ConcernMorph views: (A) Concern Metrics and (B) Crosscutting Patterns

ConcernMorph also allows users to investigate specific parts of a crosscutting pattern instance. For instance, the tool records which components are playing the role of the body and tentacles in an Octopus concern. This information can be observed by double-clicking a concern in the Crosscutting Patterns view. After this action is performed, a new window shows detailed information about the chosen concern. Figure 5-16 illustrates the relevant parts of three crosscutting patterns instances: Octopus, Tree Root, and Neural Network. For instance, we see in this figure that one component is playing the role of the Octopus' body while other six components are classified as tentacles.

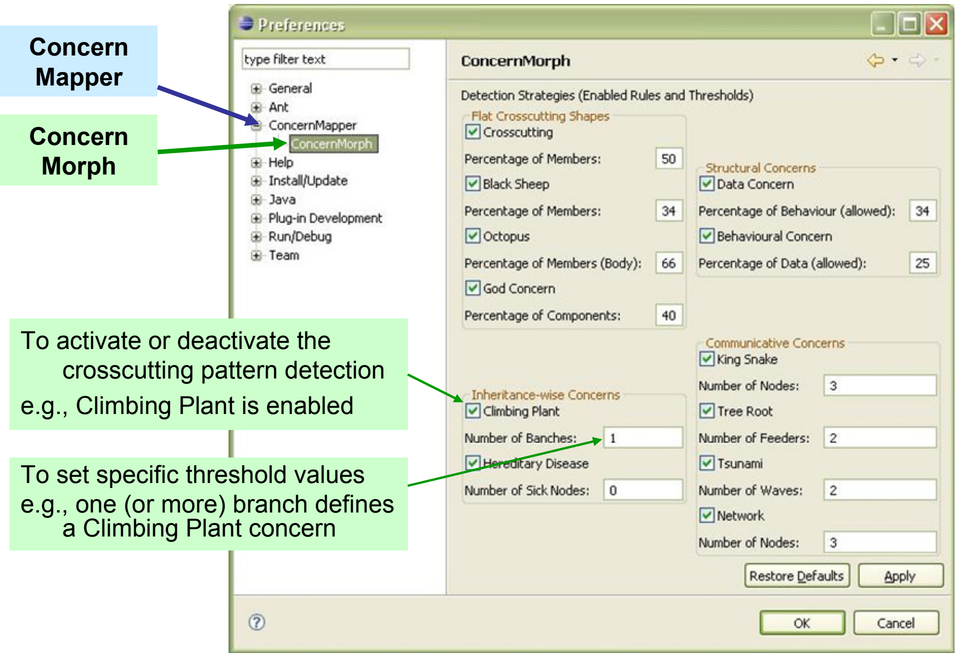

Concern Metaphors - Details
Relevant members of a crosscutting pattern are indicated
Save
Cancel

Crosscutting Patterns

Component	Octopus	God Concern	Structure	Inheritance-wise	King Snake	Tree Root	Tsunami	Network
XmlSqlMapClientBuilder	Tentacle					Feeder		Node
ResultLoader						Feeder		Node
LazyResultLoader						Feeder		Node
SqlMapSessionImpl	Tentacle					Feeder		Node
SqlMapExecutorDelegate	Tentacle					Trunk		Node
SqlMapClientImpl	Tentacle					Feeder		Node
ExtendedSqlMapClient	Body					Feeder		Node
BaseStatement	Tentacle					Feeder		Node
EnhancedLazyResultLoader						Feeder		Node
BasicResultMap	Tentacle							

Figure 5-16: Relevant parts of a crosscutting pattern in ConcernMorph

ConcernMorph extends ConcernMapper with an additional preference page (Figure 5-17). The ConcernMorph preference page supports the configuration of specific threshold values for the program under analysis. This page also allows the user to activate or deactivate the heuristic rule of a particular crosscutting pattern. For instance, Figure 5-17 shows that all crosscutting patterns are active in this specific project. In particular, according to the used preferences, a concern is classified as Climbing Plant if one or more branches are found in the crosscutting pattern structure as presented in Figure 5-17. Appendix G summarises the installation steps of ConcernMorph.



Concern Mapper

Concern Morph

To activate or deactivate the crosscutting pattern detection e.g., Climbing Plant is enabled

To set specific threshold values e.g., one (or more) branch defines a Climbing Plant concern

ConcernMorph

Detection Strategies (Enabled Rules and Thresholds)

Flat Crosscutting Shapes
☑ Crosscutting
Percentage of Members: 50
☑ Black Sheep
Percentage of Members: 34
☑ Octopus
Percentage of Members (Body): 66
☑ God Concern
Percentage of Components: 40

Structural Concerns
☑ Data Concern
Percentage of Behaviour (allowed): 34
☑ Behavioural Concern
Percentage of Data (allowed): 25

Inheritance-wise Concerns
☑ Climbing Plant
Number of Banches: 1
☑ Hereditary Disease
Number of Sick Nodes: 0

Communicative Concerns
☑ King Snake
Number of Nodes: 3
☑ Tree Root
Number of Feeders: 2
☑ Tsunami
Number of Waves: 2
☑ Network
Number of Nodes: 3

Figure 5-17: Setting thresholds in ConcernMorph

5.7 Summary

This chapter presented a suite of concern-sensitive detection strategies and algorithms for identifying crosscutting concerns and classifying them according to our crosscutting patterns. The proposed suite can be extended and, by no means, have we claimed that it is complete. For instance, new crosscutting patterns (and detecting strategies for them) can be further defined. More accurate detection strategies and algorithms can also be proposed, for instance, to better reflect the intention of the formal definitions of the crosscutting patterns presented in Chapter 4. This chapter also presented a tool, called ConcernMorph, to support the detection of crosscutting patterns. This tool was implemented as an Eclipse plugin and offers extension points. For instance, new heuristic rules can be incorporated to ConcernMorph based on these extension points.

The next chapter presents a controlled experiment with the goal of assessing the manual identification of concerns by developers. Students and young researchers from three institutions are used as subjects in this experiment. It also assesses the accuracy of the heuristic technique proposed in this chapter for detecting crosscutting concerns. Moreover, the next chapter investigates the hypothesis that concern-sensitive detection strategies offers enhancements over traditional metrics-based assessment approaches. The correlation between certain types of crosscutting patterns and design stability is investigated in Chapters 7 and 8.

6. Preliminary Analysis of Crosscutting Concerns

The main purpose of this chapter, together with Chapters 7 and 8, is to evaluate the correlation of crosscutting patterns and design stability. However, we need two preliminary evaluation steps to achieve this main goal: (i) to assess the accuracy of concern projection into software artefacts and (ii) to assess the accuracy of crosscutting concern identification. The accuracy of concern projection is questioned by many researchers [9, 26, 130]. For this reason, we first evaluate the ability of developers to accurately project concerns into software artefacts. We perform three replications of a specific controlled experiment in this first evaluation step. The controlled experiment requires students to project 6 concerns into implementation artefacts of the HealthWatcher system.

This chapter also reports a systematic evaluation of the accuracy of the heuristic detection technique to identify crosscutting concerns. This evaluation step focuses on the basic detection strategies for crosscutting concern identification presented in Section 5.1. The detection strategies and algorithms for the detection of crosscutting patterns are discussed in Chapters 7 and 8. Six systems from different application domains are used in this evaluation step. We analyse both object-oriented and aspect-oriented designs of such systems, which encompass heterogeneous forms of crosscutting concerns. The accuracy of the heuristic technique is evaluated by comparing its heuristically obtained results with previously available information about the analysed concerns.

Section 6.1 briefly presents the evaluation methodology of this chapter. Section 6.2 reports the results of the controlled experiment which evaluates accuracy of concern projection. Section 6.3 evaluates the basic detection strategies for crosscutting concern identification. Section 6.4 discusses some constraints of this evaluation and Section 6.5 summarises the results of this chapter.

6.1 Evaluation Methodology and Hypotheses

This section summarises the hypotheses and evaluation methodology of this chapter. The evaluation focuses on two basic challenges of concern analysis, namely: (i) the manual projection of concerns into software artefacts and (ii) the heuristic identification of crosscutting concerns. We describe below the hypotheses and evaluation procedures we followed to address these two research challenges.

Manual Projection of Concerns. To evaluate the impact of concern projection on the proposed heuristic assessment technique, we performed a controlled experiment (Section 6.2). This experiment investigated the *concept assignment problem* [9, 130] with the goal of verifying the hypothesis below. The *concept assignment* problem is called *concern projection* problem in the context of this thesis. This problem is a pre-requirement to measure and heuristically classify the crosscutting concerns. That is, we rely on previous assignments of concerns to the software artefacts before applying our approach.

> ***Hypothesis 6-1****: The projection of system concerns into implementation artefacts does not depend on individual differences between developers.*

To test this hypothesis, we relied on a controlled experiment which collected a set of projections for six concerns. We asked the experiment participants to project 6 concerns into software artefacts. A set of 28 subjects from 3 different institutions took part in this experiment. However, some participants of the experiment have not projected all concerns into the system artefacts (Section 6.2). We used artefacts of the Health Watcher system in this experiment and, considering data from all subjects, we collected 121 concern projections in total.

Heuristic Identification of Crosscutting Concerns. We also evaluated the accuracy and applicability of the heuristic assessment technique for identifying crosscutting concerns. In this particular case, we aimed to verify the following hypothesis.

> ***Hypothesis 6-2****: Heuristic assessment techniques which are based on concern properties enhance traditional quantitative analysis.*

To test this hypothesis, we followed three steps detailed in Section 6.3. First, we selected 23 concerns projected into design and implementation artefacts of six systems. Second, we used the concern-sensitive detection strategies presented in Section 5.1 to heuristically identify crosscutting concerns. In addition to identify crosscutting concerns, the motivation of this evaluation was to verify if concern-sensitive detection strategies minimise the shortcomings of traditional metrics-based strategies (discussed in the literature review, Section 2.5). Our hypothesis is that most of the problems of traditional strategies can be significantly ameliorated through the application of concern-sensitive analysis. Third, we evaluated whether the heuristic classification of crosscutting concerns indicates design flaws not reported by traditional software metrics. Additionally, we evaluated the accuracy of the heuristic identification of crosscutting concerns by comparing it to previous knowledge about the analysed concerns.

6.2 Accuracy of Manual Concern Projection

Before we evaluate the heuristic assessment technique to detect and classify crosscutting concerns, we report in this section the results of a controlled experiment which investigates the accuracy of manual concern identification. The goal of this experiment is to verify the Hypothesis 6-1 presented in Section 6.1. To test this hypothesis, we designed and conducted a controlled experiment based on the *concern projection problem* [9, 130] (Section 6.2.1). The following sections report our experimental procedures (Section 6.2.1), collected data (Sections 6.2.2 to 6.2.4), and generalisation of the key results (Section 6.2.5).

6.2.1 Experimental Design

People understand a program because they are able to relate the structure and environment of the program to their human-oriented conceptual knowledge about the world [9]. The problem of discovering human concepts (i.e., concerns) and assigning them to their implementation counterparts for a given program is the concern projection problem [9, 130].

The Concern Projection Problem. Concern projection involves recognising high-level concerns in the application domain of a software system and associating these concerns with artefacts at the design and implementation levels [9, 130]. Automated concern projection techniques form the foundation for a number of software engineering techniques, including feature location [151], concern mining [107, 129], and requirements traceability [45, 71, 126].

Selection of the System Concerns. To investigate the impact of the concern projection problem on our study, we designed and conducted a controlled experiment.

As part of this experiment, we selected six different concerns from Health Watcher (Section 8.1.2) and, for each concern, we asked several different subjects to produce a concern projection onto a subset of the application classes. The projection of concerns was performed by hand; i.e., subjects were not allowed to use any tool. We have not used tool support in our experiment to avoid manipulating many experimental variables, such as the ability of subjects to use such tools and the configuration and performance of different machines. Besides, we do not aim to assess any specific tool in this experiment.

Table 6-1: Crosscutting and non-crosscutting concerns analysed in Health Watcher

Concern	What they mean
Business	It defines the business elements and rules. In the Health Watcher case, Business is related to the kind of manipulated information, such as employees, complaints, and health units.
Concurrency	It provides a control for avoiding inconsistent information stored in the system's database. Concurrency control can be implemented, for example, by defining atomic database transactions.
Distribution	It is responsible for externalising the system services at the server side and supporting their distribution to the clients. Remote Method Invocation (RMI) is used by client classes to call the Health Watcher services.
Exception Handling	It is the policy used to represent exceptional conditions and to handle their occurrences. Many programming languages, such as Java, have built-in support for exceptions and exception handling. The scope for exception handlers starts with a marker clause (*try*) and ends in the start of the handler clauses (*catch*) in Java.
Persistence	It is responsible for retrieving and storing the information manipulated by the system. Database connections are used in the Health Watcher implementation of Persistence before retrieving data from and storing data into disk.
View (GUI)	It provides a Web interface that allows users to interact with the Heath Watcher system. View is implemented by servlets in Health Watcher.

The set of concerns selected in this experiment includes Business, Concurrency, Distribution, Exception Handling (EH), Persistence, and View. These concerns are briefly described in Table 6-1. The subjects were given the source code of four of the Health Watcher classes: `EmployeeRecord`, `Employee`, `HealthWatcherFacade`, and `ServletInsertEmployee`. We restricted our experiment to these four classes and six concerns because we wanted to be able to complete the experiment in a 1-hour class. Timing was constrained by the host institutions. The aforementioned classes were selected because they are representative classes of different Health Watcher layers (see Figure 8-2).

Dependent and Independent Variables. Given the exploratory nature of our experiment, we focused on the simplicity of its design to facilitate the interpretation of the results. For this reason, we decided to manipulate only three independent variables: concerns, subjects, and institutions. The *concern* variable takes as value one of the six different concerns identified by the subjects. The *subject* variable identifies one of the 23 different subjects who took part in our study. Some subjects represented more than one person because we used groups of two or three people as a single subject in the last institution (Section 6.2.4). The *institution* variable identifies (i) the background of each subject acquired by a questionnaire and (ii) the way (alone or in groups) subjects performed the experiment. We verified that subjects of the same institution have similar backgrounds in terms of aspect-oriented programming and class diagrams.

Given a concern *con*, a subject *sub*, and an institution *aff*, our only dependent variable is the mapping *projection(con, sub, aff)*. The *projection* dependent variable consists of the lines of code which the subject *sub* (affiliated to the institution *aff*) projected the concern *con* onto. To facilitate the data analysis, we defined three metrics (Hits, False Positives, and False Negatives) that abstract specific characteristics of the concern projections. The Hits metric quantifies the cases where subjects (i) correctly tagged a line of code for a concern or (ii) did not tag a line of code which was not supposed to be tagged. The False Positives (FP) metric quantifies the cases where subjects wrongly tagged lines of code for a concern. And, the False Negatives (FN) metric indicates the cases where subjects did not tag a line of code for a concern which was supposed to be tagged. The decision of whether a line of code was correctly or incorrectly tagged is based on an *oracle* of concern projection. This oracle represents the projection of concerns performed by the developers or by otherwise specialists of this application. These specialists also made their concern projections available (see Section 8.1.2).

The three previously described metrics are expressed as follows.

$$\text{Hits} = \frac{\text{\# LOC correctly tagged + \# LOC correctly not tagged}}{\text{\# Lines of Code}}$$

$$\text{FP} = \frac{\text{\# LOC wrongly tagged}}{\text{\# LOC not implementing concern}} \qquad \text{FN} = \frac{\text{\# LOC wrongly not tagged}}{\text{\# LOC implementing concern}}$$

Subject Training Session. Before running the actual experiment, the subjects were given a short 15-minute demonstration of the goals and steps to be followed. This demonstration worked as a training session since most of the subjects were not familiar with concern projection techniques. In this session, we demonstrated the projection of one concern of MobileMedia (Section 7.1) into one of its classes. We used a different application and a different concern to avoid biasing the experiment result. The training session also included some guidelines of concern projection, e.g., it indicated that different concerns could be tagged to the same code fragment and that the whole line of code should be tagged (not only part of it). At the end of the training session, subjects were given (i) the textual description of the Health Watcher system and its 6 concerns of interest and (ii) a design model of the system. The description of the concerns and the Health Watcher design model are similar to Table 6-1 and Figure 8-2, respectively.

6.2.2 Results of the First Institution

The first run of our controlled experiment was initially intended to serve as a pilot study, due to the usual problems in the first experimental design that are difficult to anticipate. Even though we made a minor adjustment of the experimental design (discussed below) in the next run of this study, no major issue emerged during the experiment execution. As a consequence, we also took the data from this study into consideration. In this case, we run the experiment with six subjects from a research centre (we call FRB) in Bahia, a northeast state of Brazil. All subjects were interns or young researchers with less than 3 years experience in software development.

By answering a questionnaire summarised in Table 6-2, the participants of this experiment claimed to have low to medium experience with class diagrams and medium to extensive experience with the Java programming language. Table 6-2 shows a summary of the subjects' backgrounds for this first institution. The columns, labelled S1 to S6, represent the six subjects of this experiment. Lines present information about the subjects and their levels of experience. For instance, Subject S1 is male and performed in his academic life 5 courses out of the 12 options. He has less than six months of working experience and considers himself moderately familiar with class diagrams and Java programming. The data of this table show that the subjects hold similar experience in class diagrams (medium to extensive) and Java programming (low to medium). They also took similar courses in their academic lives.

Table 6-2: Background information about the subjects (FRB)

Subjects		**S1**	**S2**	**S3**	**S4**	**S5**	**S6**
Sex		Male	Male	Male	Male	Male	Male
Age		20	19	19	22	22	27
Courses	Java Programming	X	X	X	X	X	X
	OO Programming	X	X	X	X	X	X
	Web Technologies	X	X		X	X	X
	Software Engineering	X	X	X	X	X	X
	Software Design						X
	UML Design						X
	Database Systems	X	X		X	X	X
	CS Innovation						
	Distributed System					X	X
	Project Management						X
	Advanced DB Tech.						
	Component Systems						
Working experience		0-6 months	0-6 months	0	0.5-1 year	1-3 years	0.5-1 year
Class diagram (skill)		medium	low	Low	low	medium	medium
Java (skill)		medium	medium	Medium	extensive	extensive	extensive

In the experimental procedures, the six subjects were asked to individually project each of the six aforementioned concerns onto four classes of Health Watcher. They were also instructed to complete this task in one hour and the time was controlled by the experiment instructor. The instructor also asked the subjects to record the amount of time spent to analyse the system and to actually project the concerns. Table 6-3 summarises the results of the experiment for the first institution. The first two lines present the time spent in the two activities: system analysis and concern projection. The remaining lines of Table 6-3 indicate the measures of Hits, False Positives, and False Negatives for each concern per subject. Each cell shows the absolute number of lines of code and its percentage according to the corresponding metric expression (see Section 6.2.1).

Table 6-3: Summary of results for concern identification (FRB)

		S1	S2	S3	S4	S5	S6	Avg
Time	System and Concern Descriptions	9 min	10 min	11 min	15 min	9 min	11 min	11 min
	Concern Identification	40 min	40 min	46 min *	43 min	42 min	35 min	41 min
Business	Hits	173/286 60.5%	171/286 59.8%	158/286 55.2%	227/286 79.4%	130/286 45.5%	204/286 71.3%	177/286 61.9%
	False Positives	0/81 0.0%	0/81 0.0%	0/81 0.0%	4/81 4.9%	0/81 0.0%	0/81 0.0%	0.7/81 0.8%
	False Negatives	113/205 55.1%	115/205 56.1%	128/205 62.4%	55/205 26.8%	156/205 76.1%	82/205 40.0%	108/205 52.8%
Concurrency	Hits	282/286 98.6%	285/286 99.7%	275/286 96.2%	286/286 100.0%	283/286 99.0%	279/286 97.6%	281/286 98.5%
	False Positives	3/281 1.1%	0/281 0.0%	9/281 3.2%	0/281 0.0%	2/281 0.7%	6/281 2.1%	3.3/281 1.2%
	False Negatives	1/5 20.0%	1/5 20.0%	2/5 40.0%	0/5 0.0%	1/5 20.0%	1/5 20.0%	1/5 20.0%
Distribution	Hits	223/286 78.0%	267/286 93.4%	267/286 93.4%	254/286 88.8%	256/286 89.5%	253/286 88.5%	253/286 88.6%
	False Positives	10/233 4.3%	2/233 0.9%	5/233 2.1%	14/233 6.0%	6/233 2.6%	3/233 1.3%	6/233 2.9%
	False Negatives	53/53 100.0%	17/53 32.1%	14/53 26.4%	18/53 34.0%	22/53 41.5%	30/53 56.6%	25/53 48.4%
EH	Hits	264/286 92.3%	263/286 92.0%	266/286 93.0%	270/286 94.4%	254/286 88.8%	283/286 99.0%	266/286 93.2%
	False Positives	20/242 8.3%	0/242 0.0%	13/242 5.4%	1/242 0.4%	13/242 5.4%	0/242 0.0%	8/242 3.2%
	False Negatives	2/44 4.5%	23/44 52.3%	7/44 15.9%	15/44 34.1%	19/44 43.2%	3/44 6.8%	11/44 26.1%
Persistence	Hits	196/286 68.5%	267/286 93.4%	280/286 97.9%	279/286 97.6%	275/286 96.2%	267/286 93.4%	260/286 91.1%
	False Positives	89/249 35.7%	10/249 4.0%	4/249 1.6%	0/249 0.0%	1/249 0.4%	3/249 1.2%	18/249 7.2%
	False Negatives	1/37 2.7%	9/37 24.3%	2/37 5.4%	7/37 18.9%	10/37 27.0%	16/37 43.2%	7.5/37 20.3%
View	Hits	265/286 92.7%	267/286 93.4%	248/286 86.7%	273/286 95.5%	234/286 81.8%	271/286 94.8%	259/286 90.8%
	False Positives	0/248 0.0%	0/248 0.0%	0/248 0.0%	0/248 0.0%	14/248 5.6%	0/248 0.0%	2/248 0.9%
	False Negatives	21/38 55.3%	19/38 50.0%	38/38 100.0%	13/38 34.2%	38/38 100.0%	15/38 39.5%	24/38 63.2%

** Subject 3 claimed that he was unable to complete the exercise in the allocated time frame (1 hour).*

The results of this pilot experiment indicate that our time frame seems to be too short for the smooth execution of the given tasks. This conclusion is supported by the observation that the third subject (S3) was unable to finish his tasks in the allocated time frame. As a result, we decided to change the experiment design by removing one of the concerns (Business) for the next two replications (Sections 6.2.3 and 6.2.4). We chose to remove Business because we felt this concern took a lot of time to be identified since it involves many lines of code. Additionally, based on post analysis of the pilot results, we realised that subjects could not properly understand this concern. For instance, as presented in Table 6-3, the percentage of Hits is usually higher than 80% for all concerns, except Business. On the other hand, the percentage of Hits for the Business concern is around 60% in average and never higher than 80%. This observation may indicate that the projection of some concerns, such as Business, is harder than the projection of others. We further discuss this issue and other observations in Section 6.2.5.

6.2.3 Results of the Second Institution

The second institution at which we run the experiment is Lancaster University (Lancs). In this case, the subjects were thirteen undergraduate students in their final year of study. The steps followed in this experiment were basically the same as the first institution (Section 6.2.2). The only difference was that we removed the Business concern as explained before. Tables 6-4 and 6-5 present the background information for the subjects 1 to 7 and for the subjects 8 to 13, respectively. The structure of these tables is the same as Table 6-2. By comparing the subjects' background of the first institution (Table 6-2) with the second institution (Tables 6-4 and 6-5), we can observe that all subjects so far had similar experiences. In the second institution, we could find just two outliers (S2 and S10) that claimed to have extensive experience with both class diagrams and Java programming. Apart from these two subjects, the others had low to medium experience with class diagrams and some experience with Java programming. They all also took similar courses, although with slightly different names since they had different affiliations. For instance, Tables 6-2, 6-4, and 6-5 show that all subjects claimed to have attended the basic Computer Science courses, such as Object-Oriented Programming and Software Engineering.

Table 6-4: Background information about the subjects 1 to 7 (Lancs)

Subjects		**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
Sex		Male	Male	Male	Male	Male	Female	Male
Age		20	20	21	20	21	20	19
Courses	CSc110: Java Progr.	X	X	X	X	X	X	X
	CSc112: OO Progr.	X	X	X	X	X	X	X
	CSc150: Web Tech.	X	X	X	X	X	X	X
	CSc160: Soft. Eng.	X	X	X	X	X	X	X
	CSc240: Soft. Design	X	X	X	X	X	X	X
	CSc242: Soft. Eng.	X	X	X	X	X	X	X
	CSc243: DB Systems	X	X	X	X	X	X	X
	CSc245: CS Innov.							
	CSc253: Distr. Sys.	X	X	X		X		
	CSc362: Proj. Man.		X				X	
	CSc364: Adv. DB							
	CSc367: Comp. Sys.		X					X
Working experience		No	No	0-6 months	No	No	No	No
Class diagram (skill)		low	extensive	low	medium	medium	low	medium
Java (skill)		medium	extensive	medium	low	medium	low	medium

Table 6-5: Background information about the subjects 8 to 13 (Lancs)

		S8	**S9**	**S10**	**S11**	**S12**	**S13**
Sex		Male	Male	Male	Male	Male	Male
Age		19	20	21	19	20	21
Courses	CSc110: Java Progr.	X	X	X	X	X	X
	CSc112: OO Progr.	X	X	X	X	X	X
	CSc150: Web Tech.	X	X	X	X	X	X
	CSc160: Soft. Eng.	X	X	X	X	X	X
	CSc240: Soft. Design	X	X	X	X	X	X
	CSc242: Soft. Eng.	X	X	X	X	X	X
	CSc243: DB Systems	X	X	X	X	X	X
	CSc245: CS Innov.						
	CSc253: Distr. Sys.			X			
	CSc362: Proj. Man.						
	CSc364: Adv. DB						
	CSc367: Comp. Sys.			X			
Working experience		No	No	0-6 months	No	No	No
Class diagram (skill)		medium	medium	extensive	medium	low	medium
Java (skill)		medium	extensive	extensive	medium	medium	medium

Table 6-6: Summary of results for concern identification (Subjects 1 to 7, Lancs)

		S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
Time	System and Concern Descriptions	3 min	4 min	4 min	13 min	5 min	5 min	6 min
	Concern Identification	14 min	28 min	24 min	18 min	21 min	12 min	23 min
Concurrency	Hits	246/246 100.0%	240/246 97.6%	246/246 100.0%	242/246 98.4%	246/246 100.0%	242/246 98.4%	240/246 97.6%
	False Positives	0/241 0.0%	6/241 2.5%	0/241 0.0%	0/241 0.0%	0/241 0.0%	3/241 1.2%	6/241 2.5%
	False Negatives	0/5 0.0%	0/5 0.0%	0/5 0.0%	5/5 80.0%	0/5 0.0%	1/5 20.0%	0/5 0.0%
Distribution	Hits	220/246 89.4%	220/246 89.4%	228/246 92.7%	197/246 80.1%	215/246 87.4%	213/246 86.6%	204/246 82.9%
	False Positives	12/193 6.2%	9/193 4.7%	9/193 4.7%	0/193 0.0%	12/193 6.2%	5/193 2.6%	29/193 15.0%
	False Negatives	14/53 26.4%	17/53 32.1%	9/53 17.0%	49/53 92.5%	19/53 35.8%	28/53 52.8%	13/53 24.5%
EH	Hits	232/246 94.3%	233/246 94.7%	217/246 88.2%	208/246 84.6%	217/246 88.2%	212/246 86.2%	243/246 98.8%
	False Positives	2/202 1.0%	13/202 6.4%	24/202 11.9%	28/202 13.9%	13/202 6.4%	19/202 9.4%	0/202 0%
	False Negatives	12/44 27.3%	0/44 0.0%	6/44 13.6%	10/44 22.7%	16/44 36.4%	15/44 34.1%	3/44 6.8%
Persistence	Hits	215/246 87.4%	216/246 87.8%	232/246 94.3%	243/246 98.8%	244/246 99.2%	239/246 97.2%	210/246 85.4%
	False Positives	30/209 14.4%	29/209 13.9%	10/209 4.8%	0/209 0.0%	0/209 0.0%	3/209 1.4%	27/209 12.9%
	False Negatives	1/37 2.7%	1/37 2.7%	4/37 10.8%	3/37 8.1%	2/37 5.4%	4/37 10.8%	9/37 24.3%
View	Hits	222/246 90.2%	233/246 94.7%	227/246 92.3%	232/246 94.3%	244/246 99.2%	227/246 92.3%	231/246 93.9%
	False Positives	1/208 0.5%	0/208 0.0%	0/208 0.0%	0/208 0.0%	0/208 0.0%	0/208 0.0%	0/208 0.0%
	False Negatives	23/38 60.5%	13/38 34.2%	19/38 50.0%	14/38 36.8%	2/38 5.3%	19/38 50.0%	15/38 39.5%

Table 6-7: Summary of results for concern identification (Subjects 8 to 13, Lancs)

		S8	**S9**	**S10**	**S11**	**S12**	**S13**	**Avg.**
Time	System and Concern Descriptions	4 min	4 min	3 min	8 min	4 min	7 min	*5 min*
	Concern Identification	18 min	13 min	22 min	24 min	23 min	14 min	*20 min*
Concurrency	Hits	240/246 97.6%	243/246 98.8%	246/246 100.0%	236/246 95.9%	242/246 98.4%	246/246 100.0%	*243/246* *98.7%*
	False Positives	1/241 0.4%	0/241 0.0%	0/241 0.0%	8/241 3.3%	3/241 1.2%	0/241 0.0%	*2/241* *0.9%*
	False Negatives	5/5 100.0%	3/5 60.0%	0/5 0.0%	2/5 40.0%	1/5 20.0%	0/5 0.0%	*1/5* *24.6%*
Distribution	Hits	198/246 80.5%	196/246 79.7%	215/246 87.4%	199/246 80.9%	219/246 89.0%	211/246 85.8%	*210/246* *85.5%*
	False Positives	3/193 1.6%	0/193 0.0%	16/193 8.3%	3/193 1.6%	13/193 6.7%	17/193 8.8%	*10/193* *5.1%*
	False Negatives	45/53 84.9%	50/53 94.3%	15/53 28.3%	44/53 83.0%	14/53 26.4%	18/53 34.0%	*26/53* *48.6%*
EH	Hits	221/246 89.8%	214/246 87.0%	242/246 98.4%	202/246 82.1%	234/246 95.1%	224/246 91.1%	*223/246* *90.7%*
	False Positives	13/202 6.4%	14/202 6.9%	1/202 0.5%	5/202 2.5%	4/202 2.0%	22/202 10.9%	*12/202* *6.0%*
	False Negatives	12/44 27.3%	18/44 40.9%	3/44 6.8%	39/44 88.6%	8/44 18.2%	0/44 0.0%	*11/44* *24.8%*
Persistence	Hits	244/246 99.2%	236/246 95.9%	239/246 97.2%	241/246 98.0%	234/246 95.1%	211/246 85.8%	*231/246* *93.9%*
	False Positives	0/209 0.0%	8/209 3.8%	7/209 3.3%	2/209 1.0%	7/209 3.3%	35/209 16.7%	*12/209* *5.8%*
	False Negatives	2/37 5.4%	2/37 5.4%	0/37 0.0%	3/37 8.1%	5/37 13.5%	0/37 0.0%	*3/37* *7.5%*
View	Hits	212/246 86.2%	212/246 86.2%	234/246 95.1%	214/246 87.0%	225/246 91.5%	213/246 86.6%	*225/246* *91.5%*
	False Positives	2/208 1.0%	0/208 0.0%	0/208 0.0%	0/208 0.0%	4/208 1.9%	0/208 0.0%	*1/208* *0.3%*
	False Negatives	32/38 84.2%	34/38 89.5%	12/38 31.6%	32/38 84.2%	17/38 44.7%	33/38 86.8%	*20/38* *53.6%*

Tables 6-6 and 6-7 show the results for all subjects of the second institution. One first observation we could make is that, in this second experiment replication, all subjects were able to finish their tasks in less than 40 minutes. We believe that this was a direct result of the removal of the Business concern because all subjects of the pilot study spent between 45 minutes and one hour to complete their tasks. Due to this observation, we decided to keep just five concerns (Concurrency, Distribution, Exception Handling, Persistence, and View) for a third replication of our experiment.

Tables 6-6 and 6-7 also show that the identification of a concern follows uniform rates of Hits, False Positives, and False Negatives, even when performed by different subjects. These results were also similar to the ones obtained in the pilot experiment (Table 6-3). For instance, the percentage of Hits for Concurrency was higher than

95% for all 19 subjects of the two institutions. Moreover, the percentage of Hits for Persistence was higher than 85% for all subjects, except S1 in the first institution. For Distribution, Exception Handling, and View, the percentage of Hits varied between 80% and 95% with few exceptions as presented by data in Tables 6-3, 6-6, and 6-7.

These results from the first two institutions would suggest that Hypothesis 6-1 (Section 6.1) is true to some extent. However, we cannot state that this hypothesis was fully satisfied only with data of the first two institutions because all subjects so far held similar experience in the areas of interest. Therefore, we ran the experiment in a third institution carefully selected to offer subjects with backgrounds as different as possible from the previous subjects. We present the results of this third institution in the next section.

6.2.4 Results of the Third Institution

The third replication of our controlled experiment was performed at PUC-Rio, a Brazilian university in the city of Rio de Janeiro. In this case, subjects were nine post-graduate (Master and PhD) students. This third replication was required because we could not fully verify the hypothesis with data from the first two institutions. That is, the condition underlined below was not properly exercised since all subjects of the first two institutions had about the same experience, as can be verified in Tables 6-2, 6-4, and 6-5. As presented in Section 6.1, our hypothesis is that "*the projection of system concerns into implementation artefacts <u>does not depend on individual differences</u> between developers*".

To cope with this issue, this time we decided to organise the subjects in groups of two (or three) members. Therefore, we relied on stronger subjects to make sure that our hypothesis is properly tested. Moreover, subjects of this replication were expected to have superior experience than subjects of the previous two institutions because they were post-graduate students and some of them worked in software companies. Therefore, based on the procedures we followed, we expected that subjects of this third institution had advantage because (i) they had higher formation degrees and (ii) they were organised in groups. Then, our aim was to verify if they indeed would come up with more accurate concern projection.

Table 6-8 presents the summary of results for the four groups of the third experiment replication. Groups were labelled G1 to G4; the first group (G1) had three members while the others had two members. Comparing the results of this institution with the previous ones, we could observe that subjects presented superior accuracy of concern identification this time. For instance, subjects of the third institution always presented better results and they achieved average percentages of Hits above 90% for all concerns. This somehow refutes our hypothesis. That is, we empirically observed that *the projection of system concerns into implementation artefacts may depend on individual differences between developers*. The next section discusses other results of the three study replications. It also tries to generalise some of our results.

Table 6-8: Summary of results for concern identification (PUC-Rio)

		G1	G2	G3	G4	Average
Time	System and Concern Descriptions	4 min	9 min	7 min	10 min	*7.5 min*
	Concern Identification	13 min	20 min	29 min	36 min	*24.5 min*
Concurrency	Hits	280/286 97.9%	286/286 100.0%	286/286 100.0%	286/286 100.0%	*284.5/286* *99.5%*
	False Positives	2/281 0.7%	0/281 0.0%	0/281 0.0%	0/281 0.0%	*0.5/281* *0.2%*
	False Negatives	4/5 80.0%	0/5 0.0%	0/5 0.0%	0/5 0.0%	*1/5* *20.0%*
Distribution	Hits	261/286 91.3%	265/286 92.7%	265/286 92.7%	259/286 90.6%	*262.5/286* *91.8%*
	False Positives	9/233 3.9%	4/233 1.7%	6/233 2.6%	8/233 3.4%	*6.8/233* *2.9%*
	False Negatives	16/53 30.2%	17/53 32.1%	15/53 28.3%	19/53 35.8%	*16.8/53* *31.6.6%*
EH	Hits	279/286 97.6%	278/286 97.2%	283/286 99.0%	277/286 96.9%	*279.2/286* *97.6%*
	False Positives	6/242 2.5%	2/242 0.8%	0/242 0.0%	0/242 0.0%	*2/242* *0.8%*
	False Negatives	1/44 2.3%	6/44 13.6%	3/44 6.8%	9/44 20.5%	*4.8/44* *10.8%*
Persistence	Hits	246/286 86.0%	278/286 97.2%	286/286 100.0%	265/286 92.7%	*268.8/286* *94.0%*
	False Positives	38/249 15.3%	7/249 2.8%	0/249 0.0%	18/249 7.2%	*15.8/249* *6.3%*
	False Negatives	2/37 5.4%	1/37 2.7%	0/37 0.0%	3/37 8.1%	*1.5/37* *4.1%*
View	Hits	272/286 95.1%	280/286 97.9%	275/286 96.2%	253/286 88.5%	*270/286* *94.4%*
	False Positives	0/248 0.0%	0/248 0.0%	0/248 0.0%	0/248 0.0%	*0/248* *0.0%*
	False Negatives	14/38 36.8%	6/38 15.8%	11/38 28.9%	33/38 86.8%	*16/38* *42.1%*

6.2.5 Discussion and Result Generalisation

As discussed in the previous section, our hypothesis was not fully confirmed. However, we observed in our controlled experiment that the percentage of Hits is usually high for all subjects (above 80%). This observation supports the use of our technique because it provides a strong indication that concerns can usually be projected into design and implementation artefacts with high degree of precision. Moreover, as we are going to discuss in Sections 7.2 and 8.2, this thesis relies upon concern projections provided by specialists or by experienced researchers working in pairs. Hence, we believe that the concern projections used in our evaluation (Chapters 7 and 8) accurately represent good samples.

Analysis of the First and Second Replications. By analysing the results of the first two institutions, we observed that, in general, subjects scored more or less the same for every concern. For instance, all nineteen subjects of these institutions scored more than 95% of Hits for the Concurrency concern. Based on this value, Concurrency seems the easier concern to be projected into this application. On the other hand, almost all subjects scored between 80% and 90% of Hits for Distribution. Apart from Business, the Distribution concern seems the hardest one to be projected into the Health Watcher implementation, although the accuracy above 80% can also be considered a good measure of Hits for quantitative heuristic methods [95, 127]. The other three concerns presented values of Hits between the two extremes; i.e., better than Distribution and worse than Concurrency.

Business was an outlier and presented poor projection accuracy. Although it was only projected by subjects of the first institution, it may represent a category of concerns which are not easily projected into software artefacts. In fact, we believe the problem with Business is mainly because it is a God Concern (Section 4.2.3). That is, it involves too much responsibility and code. Therefore, the experiment participants were not able to fully understand all different facets of Business, such as Complaint, Health Unit, Disease, and so on. Additional experimentations relying on other God Concern instances are required to support (or not) our initial findings.

Regarding the measurements of False Positives, we verified that the absolute number (and percentage) was generally low (< 10%) for all concerns. From this result, we concluded that developers usually do not assign a line of code to a concern if they are unsure about it. On the other hand, the absolute number (and percentage) of False Negatives was generally high (> 20%) for most concerns, except for Persistence. In this case, we concluded that developers usually miss many lines of code which are supposed to be tagged for a concern. A direct implication of this observation is that the use of mining tools may help developers to find code that they would otherwise miss out. For instance, we believe that most of the false negatives might be addressed by calling the developers attention for code fragments that seem relevant for a concern. These code fragments may be discovered by means of static analysis, such as clone detection techniques [7, 18] and class references [110].

Analysis of the Third Replication. Data gathered from the third institution supported most of the findings of the first two replications. For instance, Concurrency was the most accurately projected concern and Distribution was the least one. However, subjects of the third replication always achieved better results than the other two institutions (due to the facts discussed in Section 6.2.4). Moreover, the measure of False Positives was also generally low (average < 10%) for all concerns. The number (and percentage) of False Negatives was generally high (average > 10%). The Persistence concern was an exception for the second and third replications since it presented low values of False Negatives. This result might be due to the fact that subjects in the second and third institutions had more time to perform their tasks. That is, they did not project Business and, so, they had extra time to concentrate on the other concerns.

Table 6-9 presents a comparison of the average measures of Hits, False Positives, and False Negatives for subjects of the three analysed institutions. This table also supports our three key conclusions (presented below) made in this controlled experiment.

- The accuracy of concern identification depends on individual experiences, but even less experienced developers are likely to achieve reasonable accurate concern projections.
- Developers usually do not assign a line of code to a concern if they are unsure about it (which reduces the cases of false positives).
- Developers usually miss out code fragments that are realising a concern (which leads to a high occurrence of false negatives).

Table 6-9: Summary of results for concern identification

		Average per Institution		
		FRB*	**Lancs**	**PUC-Rio**
Time	System and Concern Descriptions	11 min	9 min	7 min
	Concern Identification	41 min	20 min	24 min
Concurrency	Hits	281/286 98.5%	286/286 100.0%	284/286 99.5%
	False Positives	3.3/281 1.2%	0/281 0.0%	0.5/281 0.2%
	False Negatives	1/5 20.0%	0/5 0.0%	1/5 20.0%
Distribution	Hits	253/286 88.6%	265/286 92.7%	262.5/286 91.8%
	False Positives	6/233 2.9%	4/233 1.7%	6.8/233 2.9%
	False Negatives	25/53 48.4%	17/53 32.1%	16.8/53 31.6.6%
EH	Hits	266/286 93.2%	278/286 97.2%	279.2/286 97.6%
	False Positives	8/242 3.2%	2/242 0.8%	2/242 0.8%
	False Negatives	11/44 26.1%	6/44 13.6%	4.8/44 10.8%
Persistence	Hits	260/286 91.1%	278/286 97.2%	268.8/286 94.0%
	False Positives	18/249 7.2%	7/249 2.8%	15.8/249 6.3%
	False Negatives	7.5/37 20.3%	1/37 2.7%	1.5/37 4.1%
View	Hits	259/286 90.8%	280/286 97.9%	270/286 94.4%
	False Positives	2/248 0.9%	0/248 0.0%	0/248 0.0%
	False Negatives	24/38 63.2%	6/38 15.8%	16/38 42.1%

** Subjects also projected the Business concern.*

Our conclusions complement a previous study performed by Robillard *et al*. [130]. In their preliminary study, Robillard and his colleagues relied on a small group of developers that identified concerns using ConcernMapper [130]. Our experiment differs from Robillard's one because our goal was not the evaluation of a specific tool. In fact, we have not relied on any tool in the execution of our experiment procedures as Robillard did.

6.3 Heuristic Detection of Crosscutting Concerns

One key observation of our controlled experiment presented in the previous section is that developers can achieve reasonably accurate concern projections. Based on this result, this section discusses two properties of the concern-sensitive detection strategies in the context of six software systems (Section 6.3.1). First, Section 6.3.2

investigates the applicability of this category of detection strategies for addressing limitations of metrics. Then, the accuracy of our heuristic detection strategies is analysed in Section 6.3.3 with respect to the correct identification of crosscutting concerns in both object-oriented and aspect-oriented systems.

6.3.1 Selection of the Target Applications

This section briefly introduces the six systems used to apply and to assess the accuracy of our concern-sensitive heuristic strategies for crosscutting concern identification. Table 6-10 summarises the nature of the selected concerns in each system. Apart from the Eclipse CVS Plugin [22, 36] which is a real software project, the other systems are medium-sized academic prototypes carefully designed with modularity and stability attributes as the main drivers. These systems were selected for several reasons. First, they encompass both aspect-oriented and object-oriented implementations and their previous assessments were exclusively based on software metrics [22, 60, 68, 73]. These characteristics allowed us to evaluate whether our concern-sensitive detection strategies are helpful to enhance a design assessment based only on metrics (some limitations of traditional metrics were discussed in Section 2.5.2). Furthermore, the modularity problems in all the selected systems have been correlated with external software attributes, such as reusability [60, 68, 80], design stability [22, 73], and manifestation of ripple effects [73].

Table 6-10: Selected applications and respective concerns

Application	Nature of the Concerns	Examples of the Concerns
(1) A Design Pattern library [80]	Roles of design patterns	Director role (Builder), Handler role (Chain of Responsibility), Creator role (Factory Method), Subject and Observer roles (Observer), Colleague and Mediator roles (Mediator)
(2) OpenOrb Middleware [22]	Design patterns and their compositions	Factory Method, Observer, Facade, Singleton, Prototype, State, Interpreter, Proxy
(3) AJATO [1, 58, 59]		
(4) Eclipse CVS Plugin [22, 36]	Recurring architectural crosscutting and non-crosscutting concerns	Business, Concurrency, Distribution, Exception Handling, Persistence
(5) Health Watcher [73, 137]		
(6) Portalware [68, 120]	Domain-specific concerns	Adaptation, Autonomy, Collaboration

Another reason for selecting these systems is the heterogeneity of design concerns documented (2nd column of Table 6-10), which include widely-scoped architectural concerns, such as Persistence and Exception Handling, and more localised ones, such as some design patterns. These concerns also present different characteristics and different degrees of complexity. Moreover, the selected systems are representatives of different application domains, ranging from simple design pattern instances (1st system) to a reflective middleware system (2nd), a Web-based application (5th), and a multi-agent system (6th). Finally, such systems also serve as effective benchmarks because they involve scenarios where the "aspectisation" of certain concerns is far from trivial [22, 60, 68, 73, 80]. We took the analysed concerns from the previous studies which have also used the same systems with different assessment purposes. For this reason, we indicate these publications in the 1st column of Table 6-10 for the complete descriptions of all systems and assessed concerns.

6.3.2 Solving Measurement Shortcomings

The goal of this section is to discuss whether and how concern-sensitive detection strategies address the limitations of traditional modularity assessment. To achieve this goal, we apply the concern-sensitive detection strategies presented in Section 5.1 to all 23 selected concerns of the six target applications. Table 6-11 provides the complete results for crosscutting concern identification in the object-oriented designs. Note that, the analysed concerns (2nd column) represent roles of design patterns in the first study (the Design Pattern library) and the design patterns themselves in the second and third applications. The 3rd column shows the best means of modularisation – object-oriented (OO) or aspect-oriented (AO) – for all 23 concerns based on specialists and literature claims [22, 60, 67, 68, 73, 80]. Specialists are researchers that participated in development, maintenance, and assessment of the systems. Their opinions were acquired by asking each of them to fill in a questionnaire presented in Appendix A.

Table 6-11: Results of the concern-sensitive detection strategies in the OO systems.

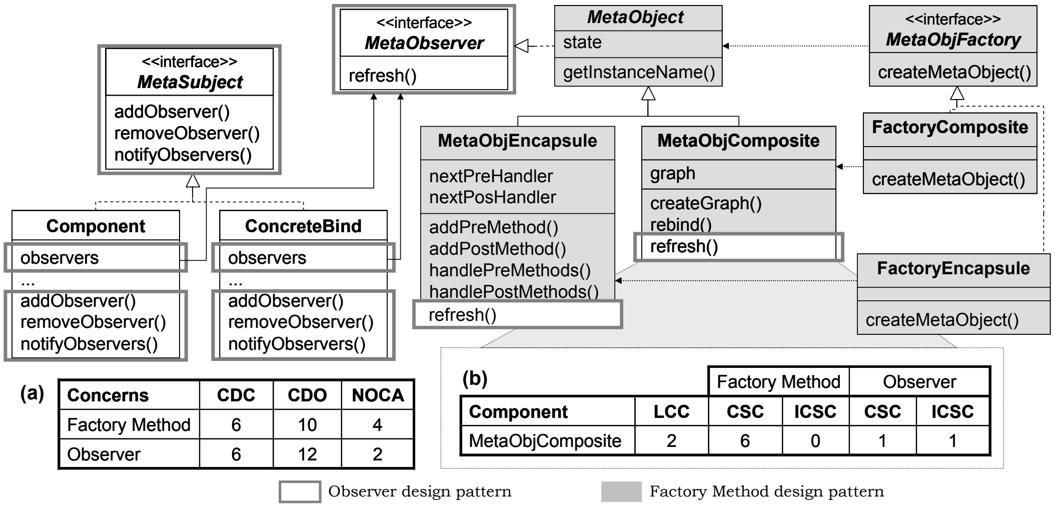
Application	Concern	Best Solution	Detection Strategies								Result
			01	02	03	04	05	06	07	08	
(1) Design Patterns Library [66, 80]	Director	OO	n	y	y	n	y	n			well-enc. / hit
	Handler	AO	n	y	n	y			y	n	well-enc. / fail
	Creator	OO	n	y	n	y			n	y	crosscutting / fail
	Observer	AO	n	y	n	y			n	y	crosscutting / hit
	Subject	AO	n	y	n	y			n	y	crosscutting / hit
	Colleague	AO	n	y	n	y			n	y	crosscutting / hit
	Mediator	AO	n	y	n	y			y	n	well-enc. / fail
(2) OpenOrb Middleware [22]	Fact. Method	OO	n	y	y	n	y	n			well-enc. / hit
	Observer	AO	n	y	y	n	n	y			crosscutting / hit
	Facade	OO	y	n							isolated / hit
	Singleton	AO	n	y	y	n	n	y			crosscutting / hit
(3) AJATO [22, 58, 59]	Prototype	AO	n	y	y	n	n	y			crosscutting / hit
	State	AO	n	y	y	n	y	n			well-enc. / fail
	Interpreter	AO	n	y	y	n	n	y			crosscutting / hit
	Proxy	AO	n	y	y	n	n	y			crosscutting / hit
(4) CVS [60]	Exc. Handl.	AO	n	y	n	y			n	y	crosscutting / hit
(5) Health Watcher [60, 73]	Business	OO	n	y	n	y			y	n	well-enc. / hit
	Concurrency	AO	n	y	n	y			n	y	crosscutting / hit
	Distribution	AO	n	y	n	y			n	y	crosscutting / hit
	Persistence	AO	n	y	n	y			n	y	crosscutting / hit
(6) Portalware [68]	Adaptation	AO	n	y	y	n	n	y			crosscutting / hit
	Autonomy	AO	n	y	y	n	n	y			crosscutting / hit
	Collaboration	AO	n	y	n	y			n	y	crosscutting / hit

The columns of Table 6-11, labelled 01 to 08, present the answers 'yes' (y) or 'no' (n) of the concern-sensitive detection strategies (rules R01 to R08 presented in Section 5.1). Blank cells mean that the rule is not applicable. Finally, the last column indicates: (i) how the set of detection strategies classifies each concern and (ii) if the classification matches with the best proposed solution ('hit') or not ('fail'). As presented before (Figure 5-1), the possible basic heuristic classification of a concern is isolated, tangled, little scattered, highly scattered, well encapsulated ('well-enc.'), and crosscutting. Based on these results, this section discusses six problems (labelled A to F) that affect not only traditional metrics but also the use of concern metrics. It also classifies these problems into four main categories. Results of the traditional and concern metrics for all systems and concerns are available in a supplementary website[6].

[6] http://www.lancs.ac.uk/postgrad/figueire/concern/heuristics/

False warnings. We identified in our analysis at least two examples of problems in this category: (A) *false scattering and tangling warnings* and (B) *false coupling or cohesion warnings*. Problem A occurs when concern metrics erroneously warn the developer of a possible crosscutting concern. However, a subsequent careful analysis shows that the concern is in fact well encapsulated. Similarly, Problem B leads developers to an unnecessary search for design flaws, but, in this situation, the false warning is caused by traditional metrics, such as coupling and cohesion ones.

Figure 6-1 presents an example of this problem category in the partial design model of the OpenOrb-complaint middleware system. This design slice focuses on elements realising the Factory Method and Observer design patterns [64]. Object-oriented abstractions provide adequate means to modularise the Factory Method since the main purpose of classes which implement this design pattern is to realise it. However, the values of the concern metrics show some level of scattering and tangling in the classes realising this pattern (tables in Figure 6-1). For instance, the CDC, CDO, and NOCA values are respectively 6, 10, and 4. These values may be considered as high as the ones obtained to the Observer concern, for instance. In this case, the false warning is a result of the Observer-specific concerns crosscutting classes of the Factory Method pattern, i.e., it is not a problem of this latter pattern implementation at all. Our studies indicate that shortcomings of this category are ameliorated with support of concern-sensitive detection strategies. For instance, Problem A discussed above (Factory Method) does not produce a false warning when the detection strategies are applied (see Table 6-11).

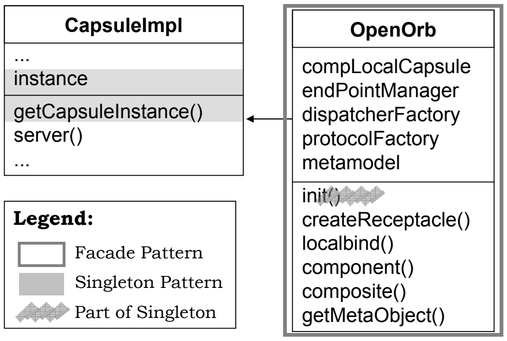


Concerns	CDC	CDO	NOCA
Factory Method	6	10	4
Observer	6	12	2

		Factory Method		Observer	
Component	LCC	CSC	ICSC	CSC	ICSC
MetaObjComposite	2	6	0	1	1

Figure 6-1: Concern metrics applied to the Observer and Factory Method patterns

Hiding concern-sensitive flaws. Sometimes, design flaws are hidden in the measurements just because (C) *metrics are not able to reveal an existing modularity problem*. We illustrate this limitation in the light of a partial class diagram presented in Figure 6-2. This figure highlights elements that implement the Singleton and Facade design patterns. It also shows some concern metrics for these patterns (tables). Although Singleton has a lower average metric value than Facade, the former presents a crosscutting nature while the latter does not. Therefore, in this example, concern metrics would probably not warn the developer about the crosscutting phenomena relative to Singleton [22, 67]. The application of concern-sensitive detection strategies overcomes this measurement shortcoming by correctly classifying the Singleton design pattern as a crosscutting concern (Table 6-11).



Pattern	CDC	NOCA	CDO
Façade	1	5	6
Singleton	2	1	1

Component	LCC
CapsuleImpl	1
OpenOrb	2

Figure 6-2: Concern metrics for Facade and Singleton.

Lack of mapping to flaw-causing concerns. Two examples of problems are classified in this category: (D) *measurement does not show where (which design elements) the problem is* and (E) *measurement does not relate design flaws to concerns causing them*. One example of Problem D is the Collaboration concern of the Portalware system which is highly scattered and tangled [68], e.g., the CDC metric is 15 (measurements can be verified at the supplementary website presented in the footnote 6 of page 120). Nevertheless, in a more detailed analysis we found out that only six components have tangling among Collaboration and other system concerns. Hence, focusing on Collaboration the developer needs to inspect (and perhaps improve) the design of only six components and not 15 as indicated by CDC. In order to solve this problem, the concern-sensitive detection strategies classified Collaboration as crosscutting (Table 6-11). Moreover, the R02 detection strategy is able to indicate the six components which have tangled concerns.

Controversial outcomes from concern metrics. A problem of this category occurs if (F) *the results of different metrics do not converge to the same outcome*, making the designer's interpretation difficult. We have identified some occurrences of this problem in our studies, like the Adaptation concern of Portalware. Applying the concern metrics to Adaptation, the CDC value is 3 (indicating low scattering) while other concern metrics present high values (e.g., NOCA=10 and CDO=22) – see the footnote 6 of page 120. Hence, the concern metrics are contradictory in the sense that it is hard to conclude whether Adaptation is a crosscutting concern. Concern-sensitive detection strategies addressed this category of problems in a number of cases. For instance, although Adaptation has contradictory results for concern metrics, the strategies have successful identified it as crosscutting (Table 6-11).

6.3.3 Accuracy of the Heuristic Detection Technique

Focusing on the basic strategies presented in Section 5.1, this section discusses the accuracy of our heuristic technique by comparing outcomes of detection strategies with the specialists' opinion or with the best known solution (3rd column of Table 6-11). We focus on the strategies for crosscutting concern identification because most of the specialists (e.g., developers who answered our questionnaire) are not familiar with the proposed crosscutting patterns. Therefore, it would be hard to verify from this perspective the correctness of the strategies and algorithms for crosscutting pattern detection.

For the analysis of the basic detection strategies, we count how many times the heuristic classification matches previous knowledge (hits) and how many times it does not match (false positives or false negatives). A false positive is a case where the concern-sensitive detection strategies answered 'yes' while the answer should have been 'no', i.e., they erroneously reported a crosscutting concern. On the other hand, a false negative is the case where the rules answered 'no' while the answer should have been 'yes'; thus, they failed to identify a crosscutting concern. In this step of our evaluation, in addition to the original object-oriented designs, we also discuss the results related to the aspect-oriented design versions. The aspect-oriented implementations aim at modularising one or more crosscutting concerns found by the developers in the object-oriented counterparts.

Table 6-12 presents the detailed results for the application of the detection strategies in the aspect-oriented systems. This table follows the same structure of Table 6-11 presented before. For instance, the first three columns of Table 6-12 have exactly the same content of their counterparts in Table 6-11. We show these columns just to facilitate the analysis of the results. Of course, since the implementations of the concerns are different (object-oriented vs. aspect-oriented versions), the detection strategies may give different results as presented by columns labelled 01 to 08 and the last column. For instance, the Creator role (Design Patterns Library) is classified as crosscutting in the object-oriented design (Table 6-11) and as well-encapsulated in the aspect-oriented one (Table 6-12).

Table 6-12: Results of the concern-sensitive detection strategies in the AO systems.

Application	Concern	Best Solution	Detection Strategies								Result
			01	02	03	04	05	06	07	08	
(1) Design Patterns Library [66, 80]	Director	OO	n	y	y	n	y	n			well-enc. / hit
	Handler	AO	n	y	y	n	y	n			well-enc. / hit
	Creator	OO	n	y	n	y			y	n	well-enc. / hit
	Observer	AO	n	y	y	n	n	y			crosscutting / fail
	Subject	AO	n	y	y	n	n	y			crosscutting / fail
	Colleague	AO	n	y	y	n	n	y			crosscutting / fail
	Mediator	AO	n	y	y	n	n	y			crosscutting / fail
(2) OpenOrb Middleware [22]	Fact. Method	OO	y	n							isolated / hit
	Observer	AO	y	n							isolated / hit
	Facade	OO	y	n							isolated / hit
	Singleton	AO	y	n							isolated / hit
(3) AJATO [22, 58, 59]	Prototype	AO	y	n							isolated / hit
	State	AO	y	n							isolated / hit
	Interpreter	AO	y	n							isolated / hit
	Proxy	AO	y	n							isolated / hit
(4) CVS [60]	Exc. Handl.	AO	n	y	n	y			n	y	crosscutting / hit *
(5) Health Watcher [60, 73]	Business	OO	n	y	n	y			y	n	well-enc. / hit
	Concurrency	AO	y	n							isolated / hit
	Distribution	AO	n	y	y	n	y	n			well-enc. / hit
	Persistence	AO	n	y	n	y			y	n	well-enc. / hit
(6) Portalware [68]	Adaptation	AO	n	y	y	n	y	n			well-enc. / hit
	Autonomy	AO	n	y	y	n	n	y			crosscutting / fail
	Collaboration	AO	y	n							isolated / hit

** Exception Handling was not totally aspectised in the CVS Plugin, following well-accepted guidelines [61]. Therefore, this concern is not a misclassification of the concern-sensitive detection strategies.*

Table 6-13 provides an overview of the hits, false positives (FP), and false negatives (FN) for the rules in the analysis of the 46 concern instances (23 in object-oriented and 23 in aspect-oriented designs). The rows of this table are organised in three parts: the object-oriented instances (OO), the aspect-oriented instances (AO), and the general data of both paradigms (OO+AO). Each row describes the absolute numbers and its percentage in relation to the total number of concerns. According to data in Table 6-13, the concern-sensitive detection strategies failed in about 20% of the cases (6 false positives and 3 false negatives). Focusing on the object-oriented designs, we found out 1 false positive and 3 false negatives. The false positive occurs with the Creator role of the Factory Method design pattern (Table 6-11). In this pattern implementation, the `GUIComponentCreator` class which plays the Creator role has also GUI-related design elements, such as the `showFrame()` method [80]. Our conclusion is that, although the Factory Method pattern is usually not a crosscutting concern [67, 80], this particular instance presents a problem of separation of concerns. Therefore, the concern-sensitive detection strategies were able to detect a problem which has not been reported by the specialists.

Table 6-13: Statistics for crosscutting concern detection

Studies	**Hits (%)**	**FP (%)**	**FN (%)**	**Total (%)**
OO	19 (82.6)	1 (4.3)	3 (13.0)	23 (100)
AO	18 (78.3)	5 (21.7)	0 (0.0)	23 (100)
OO +AO	37 (80.4)	6 (13.0)	3 (6.5)	46 (100)

Simple Program Slices. Following the same evaluation method, the detection strategies also presented 3 instances of false negatives in the object-oriented designs. These instances are: (i) the Chain of Responsibility design pattern, (ii) the Mediator role of the Mediator design pattern, and (iii) the State design pattern. The Chain of Responsibility design pattern was not detected as a crosscutting concern because the pattern instance is too simple (due to its library nature [80]) in order to expose the pattern's crosscutting nature [22, 67]. In fact, classes playing the Handler role have only three members: the `successor` attribute, the `handleClick()` method, and one constructor. Since the first two realise the Chain of Responsibility design pattern, the detection strategies wrongly classify this concern as Well-Encapsulated (i.e., the main purpose of all components). A similar situation occurs with the Mediator design pattern (Table 6-11).

Selection and Granularity of Concerns. The State design pattern is a false negative in the third application due to the assessment strategy of using patterns, instead of roles, as concerns. Although State has two roles (Context and State), it was considered as a single concern in the third application. Additionally, the state transitions (part of State role) occur in classes playing the Context role. In other words, one role crosscuts only the other role, and the concern-sensitive detection strategies were unable to indicate the pattern as a crosscutting concern. Hence, our conclusion in this case (for object-oriented designs) is that finer-grained concerns, such as roles of design patterns, would enable higher precision of the detection strategies.

Table 6-13 also presents five false positives in the heuristic assessment of aspect-oriented systems. Similarly to the State case discussed above, recurring false positives occur in aspect-oriented designs when a pattern defines more than one role. For instance, four out of five false positives match this criterion: Subject and Observer roles (Observer design pattern) and Mediator and Colleague roles (Mediator design pattern). Although these patterns are successfully modularised using abstract protocol aspects and concrete aspects, their roles share these components. Then, the Observer and Mediator roles were misclassified as crosscutting in the protocol aspect while their counterparts (Subject and Colleague) were misclassified as crosscutting in the concrete aspects. Therefore, unlike object-oriented designs, our analysis suggests selecting the whole pattern (instead of its roles) for aspect-oriented designs. In other words, the selection of the assessment granularity has to match the granularity of the aspects. So, since design patterns were aspectised, they should represent the granularity of the assessed concerns.

Aspect Precedence. The Autonomy concern is also misclassified as crosscutting in the aspect-oriented design of Portalware. This is due to a precedence definition between the Adaptation and Autonomy aspects. That is, the Adaptation aspect has a *declare precedence clause* [87] with a reference to Autonomy. So, the concern-sensitive detection strategies point out Autonomy code inside the Adaptation aspect and, therefore, indicate that the former aspect crosscuts the latter. Although this problem really exists, there is no way of improve the separation of these two concerns in the Portalware system due to AspectJ constraints [21, 73].

6.4 Threats to Validity

The participants of the controlled experiment presented in Section 6.2 are students or young researchers. Although this experiment was not compulsory for the participants, we did not expect them to be unhappy or discouraged in performing the experimental tasks. In fact, we designed and presented the experiment to the participants in a way to motivate them. Moreover, in the case of second and third institutions, the experiment execution was accounted as an extra activity aligned to the purpose of their course.

Another confounding factor in this experiment could be the subject's experience. In fact, the subjects filled in a questionnaire about their experience and expertise in academia. These data, presented in Section 6.2, were used in order to identify this possible confounding factor. The comparison between specific subjects would increase the impact of each subject's expertise in the experiment results. For this reason, rather than comparing a sample from one group with a sample from another group, we focused on the comparison of the average values from the three host institutions. Data from two specific subjects should not be directly compared to each other since they might have different expertise.

The goal of the second evaluation of this chapter (Section 6.3) was to assess how concern-sensitive detection strategies can reduce false positives and false negatives of metrics and traditional heuristic assessment techniques. We also evaluated the accuracy of the concern-sensitive detection strategies for crosscutting concern identification. Our results indicated that the concern-sensitive detection strategies presented an accuracy of about 80%. However, someone may argue that the accuracy of the detection strategies relies on the ability of developers to project concerns onto software artefacts. The reliability and feasibility of concern projection was assessed in Section 6.2.

Based on the application of concern-sensitive detection strategies to six software systems, we have shown that these strategies are accurate means for crosscutting concern identification and are usable in practice. However, we do not claim that our data are statistically relevant due to the small number of analysed systems and concerns. We have also not used rigorous statistical methods in this empirical evaluation. Nonetheless, we are considering the sample representative of the population due to the heterogeneity of systems and concerns involved in this study (Section 6.3.1).

The definition of proper threshold values intrinsically characterises any quantitative assessment approach [95, 100, 111, 131]. Our strategy was to use 50% as default for the concern-sensitive detection strategies evaluated in this chapter (and presented in Section 5.1). These values rely on our previous experience on concern analysis [52-57] and on similar values suggested by other authors [95, 111]. Of course, threshold values might need to be varied depending on application particularities and assessment goals. Our tool (Section 5.6) allows the configuration of specific threshold values for each specific system.

6.5 Summary

This chapter evaluated the reliability and feasibility of concern projection in software artefacts. It also evaluated the accuracy of crosscutting concern identification by the heuristic technique proposed in Chapter 5. To evaluate the reliability and feasibility of concern projection, we performed a controlled experiment involving 28 students and researchers from three different institutions (Section 6.2). We found out that some concerns, such as Concurrency, are easily projected by the subjects while others, such as Business, may be hard to be accurately projected into design artefacts. The results of our controlled experiment also demonstrated that most concerns can be projected onto implementation artefacts with more than 80% precision. This observation favours the practical application of concern-oriented assessment techniques.

Based on the results of the controlled experiment, this chapter also investigated the hypothesis that concern-sensitive detection strategies offer enhancements over typical metrics-based assessment approaches (Section 6.3.2). Our investigation indicated promising results in favour of concern-sensitive detection strategies for determining concerns that should be aspectised. The results pointed out that our heuristic analysis has around 80% precision for identifying crosscutting concerns compared to previous knowledge about the analysed concerns (Section 6.3.3). Additionally, detection strategies provided enhancements both for purely object-oriented and for aspect-oriented design assessments.

However, we also identified some problems in the accuracy of the proposed concern-sensitive detection strategies when (Section 6.3.3): (i) they were applied to small design slices (e.g., the Design Pattern library [80]), and (ii) there was a presence of concern overlapping. Similar problems remained when the detection strategies were applied to the corresponding aspectual designs. For instance, some concerns were misclassified in scenarios involving an explicit precedence declaration between two or more aspects. However, even in the cases of these intricate concern interactions (all systems, except the Design Patterns library) the proposed heuristic analysis presented acceptable rates (Table 6-13). Of course, this initial analysis is a stepping stone towards understanding how concerns can be useful assessment abstractions and, so, further evaluation is performed in Chapters 7 and 8. For instance, we will investigate in the next chapters the correlation between certain types of crosscutting patterns and design stability [52]. In particular, we will show in the next chapters that some crosscutting patterns seem to be more harmful to design stability than others.

7. Assessment of Architecture Stability

The cost to fix a design flaw found in early design phases is orders of magnitude less than to correct the same design flaw found during testing [31]. The system architecture is the earliest design phase that enables reasoning about critical requirements and constraints of all subsequent software refinements [6]. The occurrence of crosscutting concerns in the system architecture is associated with different maintainability impairments, such as the architecture instability [54, 133]. However, there is little knowledge about which types of crosscutting concerns are harmful to architecture stability. Therefore, it seems natural to investigate the impact of crosscutting patterns in software architecture.

This chapter reports an empirical study to investigate the correlation of crosscutting patterns and architecture stability. We use the heuristic assessment technique presented in Chapter 5 to detect instances of crosscutting patterns in the architecture models of MobileMedia (Section 7.1). MobileMedia was carefully chosen from the product line domain where the modularity of concerns plays a pivotal role. This system was previously used in Section 3.5 to empirically compare existing concern metrics. Based on the analysis of concerns in the MobileMedia architecture, this chapter aims to verify the following hypothesis.

> ***Hypothesis 7-1****: Some specific crosscutting patterns are more harmful to architecture stability than others.*

To test this hypothesis, this evaluation relies on the projection of a set of 14 concerns into the target system architecture. This set includes architecture-relevant concerns, such product-line features and non-functional requirements. Section 7.2 details the research methodology we followed. The selected concerns were projected into architecture artefacts of 8 successive releases of MobileMedia (Section 7.3). After the concerns have been projected, we measured the number of crosscutting pattern instances that could be found in the architecture models. We call this measurement *crosscutting pattern density* (Section 7.4). Similarly, the number of architecture changes over the 8 releases was calculated in order to compare this instability symptom with the crosscutting pattern density. Section 7.5 relies on this comparison to investigate the correlation of crosscutting patterns and architecture stability.

Finally, Section 7.6 discusses the experiment validity and Section 7.7 summarises this chapter.

7.1 The MobileMedia Software Product Line

The evaluation presented in this chapter relies on a software product line, called MobileMedia [56, 148], for deriving applications that manipulate photo, music, and video files on mobile devices, such as mobile phones. The current release of MobileMedia has about 4 KLOC. It evolved from a previous product line, called MobilePhoto [148, 149], developed at University of British Columbia. In fact, in order to implement MobileMedia, the developers extended the core implementation of MobilePhoto including new mandatory, optional and alternative features (Section 8.2.2).

Figure 7-1 presents a simplified view of the MobileMedia features using the feature model notation [124]. The alternative features are the types of media supported: Photo, Music, and Video. Examples of core features are: Create Media, Delete Media and Edit Label. In addition, some optional features include Transfer Photo via SMS, Sorting, Copy Media and Set/View Favourites. The core features of MobileMedia are applicable to all mobile devices that are J2ME enabled. The optional and alternative features are available for selected devices depending on the provided API support. MobileMedia was developed for a family of 4 brands of devices [148, 149], namely Nokia, Motorola, Siemens, and RIM.

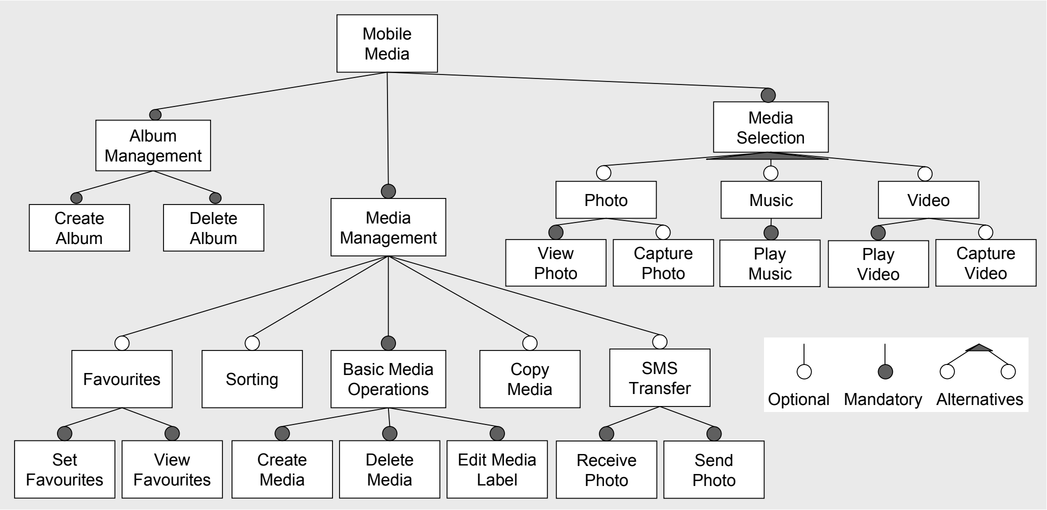


Figure 7-1: Feature model of MobileMedia

The architecture designs of MobileMedia are mainly determined by the use of the Model-View-Controller (MVC) architectural pattern [20]. Figure 7-2 presents a view of the MobileMedia architectural design using the Component & Connector notation [144]. The three grey boxes determine components that realise each of the three roles of the MVC pattern, namely model, view, and controller.

Several reasons justify the choice of MobileMedia in this study. First, it is a non-trivial system with multiple releases available, each of them introducing realistic, heterogeneous change scenarios. Second, a reasonable set of architecture artefacts and related documents is available for all releases [34, 149]. The MobileMedia design is particularly rich in several kinds of concerns, including mandatory, optional, and alternative features as well as non-functional requirements. Third, the MobileMedia architecture models were developed with modularity and maintainability principles as main driving design criteria. They were also extensively discussed in a controlled manner [34, 54, 56]. Furthermore, qualitative and quantitative studies relying on MobileMedia artefacts have been recently conducted [21, 56, 148, 149] allowing us to compare our results with these studies.

The choice of this system was also motivated by the fact that we were able to recover, document, and trace the key architectural concerns of MobileMedia together with the original and release architects. The relationship between concerns and the architectural elements were done by additional traceability documents [45]. Figure 7-3 illustrates one of these textual documents describing which elements of the MobileMedia architecture realise the Music concern. The specification of four kinds of architectural elements can be indentified in this figure: components, requires interfaces, provides interfaces, and operations. For instance, the `MediaListScreen` component has a requires interface (`ControlMedia`) which in turn has an operation (`playMusic()`) realising the Music concern.

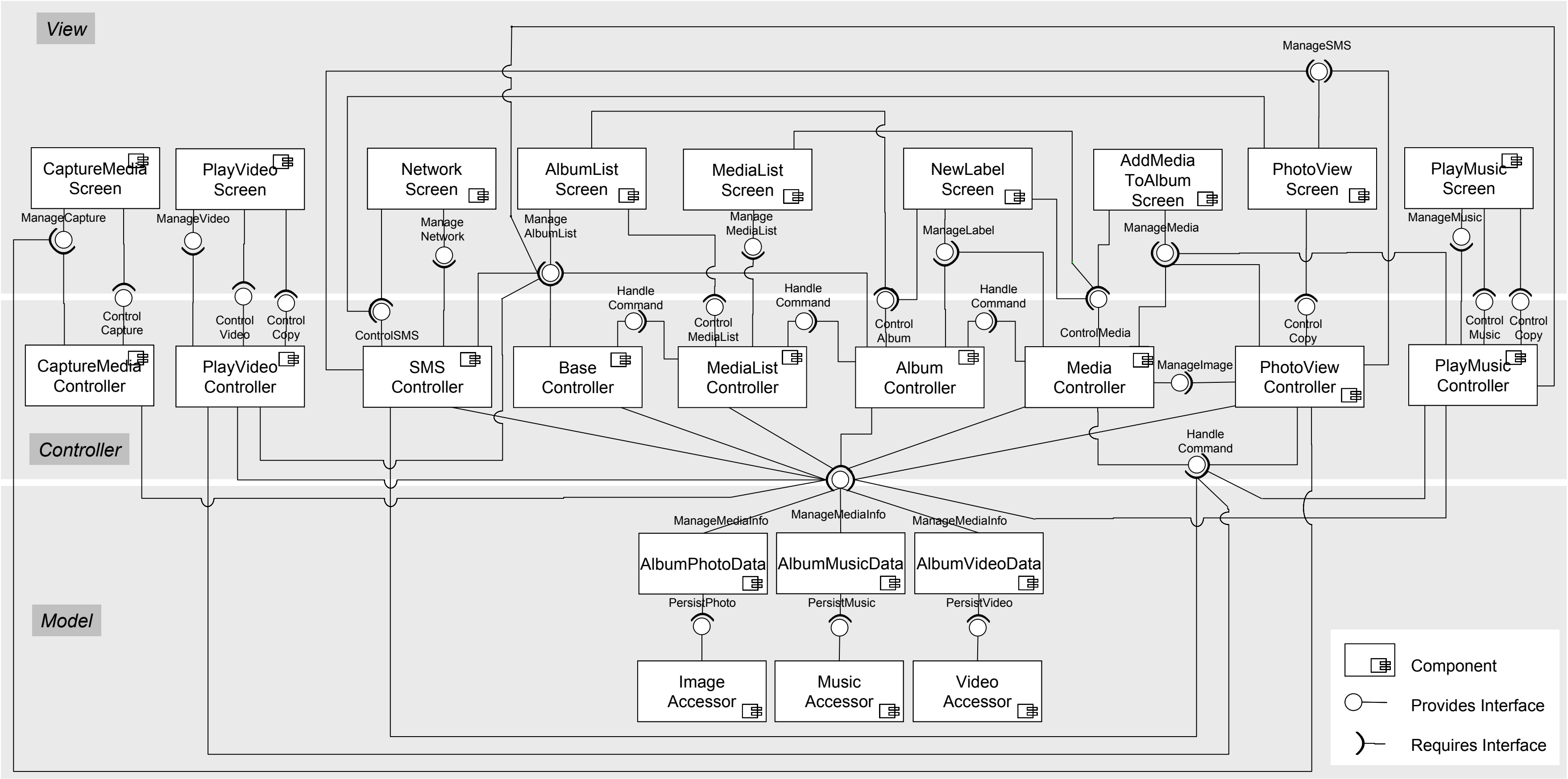
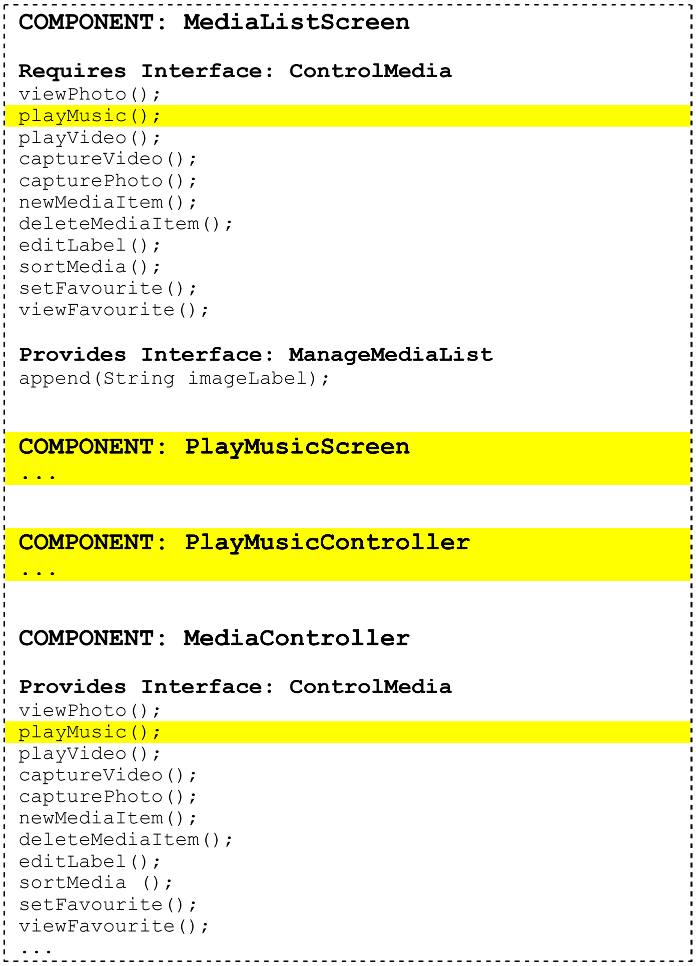


Figure 7-2: Component & Connector architecture model of MobileMedia



```
COMPONENT: MediaListScreen

Requires Interface: ControlMedia
viewPhoto();
playMusic();
playVideo();
captureVideo();
capturePhoto();
newMediaItem();
deleteMediaItem();
editLabel();
sortMedia();
setFavourite();
viewFavourite();

Provides Interface: ManageMediaList
append(String imageLabel);


COMPONENT: PlayMusicScreen
...


COMPONENT: PlayMusicController
...


COMPONENT: MediaController

Provides Interface: ControlMedia
viewPhoto();
playMusic();
playVideo();
captureVideo();
capturePhoto();
newMediaItem();
deleteMediaItem();
editLabel();
sortMedia ();
setFavourite();
viewFavourite();
...
```

Figure 7-3: Projection of the Music feature into the MobileMedia architecture

7.2 Experiment Setting Outline

This section describes the configuration of this empirical study. When setting this study, we first select a set of relevant concerns in MobileMedia (Section 7.2.1). Then, a set of change scenarios were developed for this system (Section 7.2.2). The development of new change scenarios were based on the needs of the original product line. Finally, we perform a set of experimental tasks to analyse our assessment technique with respect to architecture stability (Section 7.2.3).

7.2.1 Selection and Projection of Concerns

Six researchers worked together to select and trace 14 relevant concerns in all the releases of the MobileMedia architecture. These concerns are classified in three categories (see Section 3.2.7): *features*, *non-functional requirements*, and *roles of architectural patterns*. Features are properties in the problem domain which appear in the feature model [56]. Non-functional requirements usually do not appear in the feature model, but crosscut elements of the design realising other concerns. Architectural patterns are general reusable solutions to recurring problems in software architectural designs [32]. The representative concerns of varying and mandatory features (Figure 7-1) were selected from the original or later releases of MobileMedia. The list includes: Album, Capture, Copy, Favourites, Label, Media, Music, Photo, SMS, Sorting, and Video. Two non-functional requirements, Exception Handling and Persistence, and the Controller role of the MVC architectural pattern [32] were also considered in this study. We projected all 14 concerns in architecture artefacts of MobileMedia.

The selection and projection of concerns in this application were performed by six post-graduate researchers (and three mediators[7]) from four institutions, namely Lancaster University (Lancs), Sao Paulo University (USP), Federal University of Rio Grande do Norte (UFRN), and Federal University of Bahia (UFBA). In the concern selection step, discussion took place among all nine people involved. Then, the six researchers were organised in pairs. Each pair projected together four or five concerns into the architectural models and source code. Table 7-1 lists the set of concerns assigned to each pair of researchers. The concern-researcher assignment took the expertise of each researcher into consideration. For instance, while Researcher 1 has previous knowledge in design patterns, Researchers 3 and 5 have expertise in exception handling and persistence mechanisms, respectively.

Table 7-1: Set of MobileMedia concerns identified by each group of researchers

Pair of Researchers	Concerns
Researcher 1 (Lancs) and Researcher 2 (USP) * Mediator A (USP)	Favourites, Album, Copy, Photo, and Chain of Responsibility (Controller)
Researcher 3 (Lancs) and Researcher 4 (UFRN) * Mediator B (UFRN)	Capture, Video, Label, and Exception Handling
Researcher 5 (Lancs) and Researcher 6 (UFBA) * Mediator C (Lancs)	Media, Music, Sorting, SMS, and Persistence

[7] Mediators are the supervisors of the involved post-graduated researchers.

7.2.2 Change Scenarios

We consider seven change scenarios in the MobileMedia architecture leading to eight releases. These scenarios were generated in the last two years, following the original architectural model designed in 2005 [148, 149]. Table 7-2 summarises the changes made in each release. The scenarios comprise different types of changes involving mandatory, optional, and alternative features, as well as non-functional requirements. Table 7-2 also presents which types of architectural changes each release encompassed. The purpose of these changes is therefore to exercise the modular structure of the key concerns of MobileMedia. We rely on these change scenarios to observe the architecture stability of the product line.

Table 7-2: Summary of scenarios in MobileMedia

	Change Scenarios	**Type of Change**
0	The base MobilePhoto release [148, 149]	
1	Exception handling included (in the AspectJ version, exception handling was implemented according to [61])	Inclusion of non-functional requirements
2	New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo label	Inclusion of optional and mandatory features
3	New feature added to allow users to specify and view their favourite photos.	Inclusion of optional feature
4	New feature added to allow users to keep multiple copies of photos	Inclusion of optional feature
5	New feature added to send photo to other users by SMS	Inclusion of optional feature
6	New feature added to store, play, and organise music. The management of photo (e.g., create and delete) was turned into an alternative feature. Optional features (e.g., sorting and favourites) were provided for music	Changing of one mandatory feature into two alternatives
7	New feature added to manage videos	Inclusion of alternative feature

7.2.3 Study Outline

Aiming to test Hypothesis 7-1 presented in the beginning of this chapter, the empirical evaluation was divided in five parts as following described.

1. **Identification and Measurement of Architectural Concerns**. We selected and projected all concerns into the architectural artefacts of MobileMedia (Section 7.2.1). All concerns were then measured in order to provide input for our heuristic classification of concerns.

2. **Heuristic Detection of Crosscutting Patterns**. The classification of architectural concerns according to the crosscutting patterns was manually performed (Section 7.3). Although the classification in architecture models was not supported by our tool, it relied on the heuristic rules presented in Chapter 5.

3. **Measurement of Crosscutting Pattern Density**. Provided that all concerns were classified according to their respective crosscutting patterns, we measured the number of crosscutting pattern instances per architectural component (Section 7.4).

4. **Measurement of Component Stability**. Similarly to the previous step, we measured the number of changes per component considering the change scenarios. This step allowed us to identify the most unstable components of the target product-line architecture.

5. **Correlation of Crosscutting Patterns and Component Stability**. Finally, we compared the components with high number of crosscutting patterns and with high incidence of changes during the system evolution (Section 7.5). This step aimed to show which kinds of crosscutting patterns are good indicators of architectural changes (and which ones are not).

7.3 Heuristic Detection of Architectural Crosscutting Patterns

A concern is usually realised by a number of scattered architectural elements. The projection of architectural concerns is therefore used to document which architectural artefacts are influenced by a stakeholder's concern. This section describes how crosscutting patterns were detected based on the projection of 14 concerns onto 8 releases of the MobileMedia architecture. We documented the projection of the key MobileMedia concerns into all architectural components, interfaces, and operations by means of traceability documents as presented in Figure 7-3 (Section 7.1). The projection of concerns enabled us to identify five of the crosscutting patterns in MobileMedia architecture models, namely Black Sheep, Octopus, King Snake, Tsunami, and Tree Root. These architectural crosscutting patterns were documented based on the instantiation of our meta-model (Section 3.1) to the Component & Connector architectural models used in the architecture specification of MobileMedia. This instantiation is represented in Table 3-1 (Section 3.1.4).

Other crosscutting patterns could not be detected because the architectural models used in the MobileMedia specification do not provide enough information as required by some patterns' definitions. For instance, inheritance relationships are required to detect the Climbing Plant and Hereditary Disease crosscutting patterns. However, these relationships are not used in the Component & Connector architectural models of MobileMedia. Similarly, Copy Cat and Dolly Sheep are defined based on the notion of replicated elements which could not be easily detected in the MobileMedia architecture. Moreover, the difference between Data Concern and Behavioural Concern lies on the existence of attributes. As attributes are not defined in Component & Connector models of MobileMedia, we could not distinguish between these two crosscutting patterns. That is, all crosscutting concerns would be classified as Behavioural Concern which does not contribute to our analysis.

Figure 7-4 presents a representative partial architectural view of MobileMedia. This figure also shows the projection of four concerns (Sorting, Label, Controller, and Persistence) into architectural elements of this system. As explained in Section 7.1, the MobileMedia architecture is mainly based on the Model-View-Controller pattern [20] (grey boxes). Figure 7-4 also relates the architecture components and interfaces with the four selected MobileMedia concerns. This is done by the colours of components and provides interfaces which reflect our concern traceability documents (see Figure 7-3). If a provides interface realises a concern, all requires interfaces connected to that provides interface also realise the same concern. For instance, the Label concern is realised by the `NewLabelScreen` component, three provides interfaces (`ManageLabel`, `ManageMedia`, and `ManageImage`), and five requires interfaces connected to the provides ones.

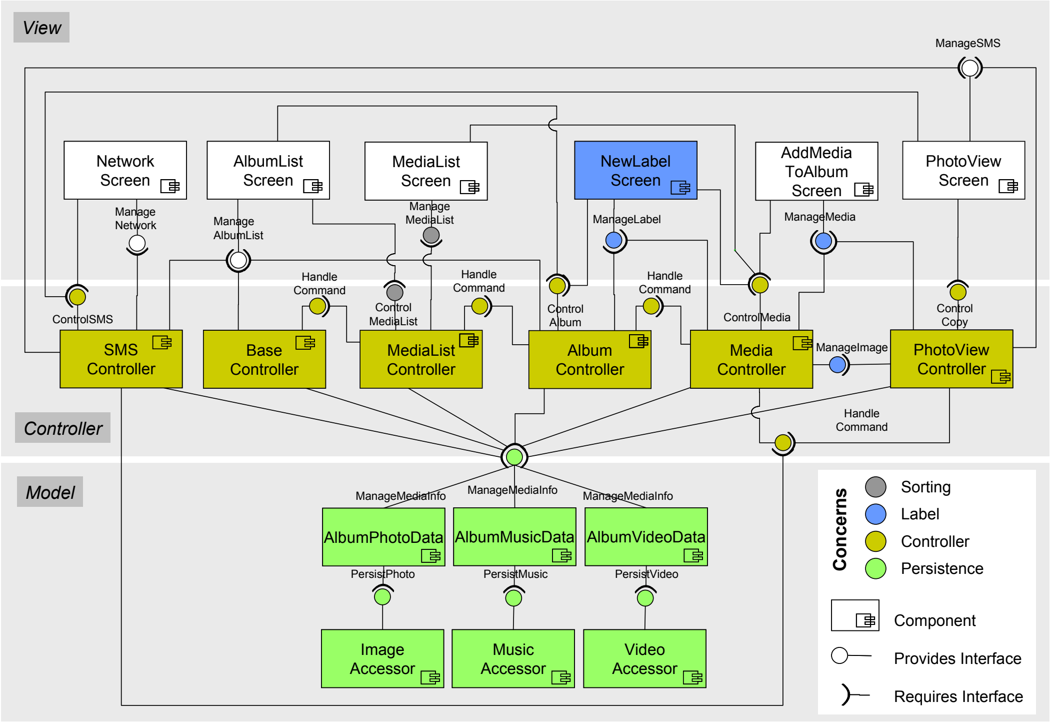


Figure 7-4: Concern projection in a partial architecture model of MobileMedia

Based on the concerns projected into the architecture model presented in Figure 7-4, we can observe instances of different architectural crosscutting patterns. For instance, this figure shows that Sorting behaves as a Black Sheep instance because only four interfaces (two provides and two requires) in two components are used to realise this concern. Similarly, Label is an Octopus concern in our example since it is well modularised by the `NewLabelScreen` component, but it also affects eight interfaces in several other components. By considering the components realising Controller, we identified a King Snake instance represented by a chain of five connected components: `PhotoViewController` (snake *head*), `MediaController`, `AlbumController`, `MediaListController`, and `BaseController` (snake *tail*). Finally, Figure 7-4 shows a Tsunami instance formed by components and interfaces realising the Persistence concern. This crosscutting pattern instance is represented by the `ManageMediaInfo` interface (*wave source*) which directly or indirectly connects (i.e., requires service) to six components realising the same concern. All instances of architectural crosscutting patterns detected in MobileMedia are documented and analysed in Sections 7.4 and 7.5.

7.4 Crosscutting Pattern Density in Architectural Models

This section reports the number of crosscutting pattern instances detected in the architectural models of MobileMedia. It analyses all 14 concerns (Section 7.2.1) based on five crosscutting patterns (Section 7.3) in all 8 releases of the MobileMedia architecture. Figure 7-5 presents charts with the results which indicate the number of components participating in the structure of each analysed crosscutting pattern. We focus on a subset of representative concerns, but the complete data are presented in Appendix B. The x-axis of each chart indicates the releases while the y-axis depicts the number of components forming the corresponding crosscutting pattern. Some concerns, such as Sorting and Copy, were not implemented in earlier releases. Data presented in these charts allow us to identify two distinct situations involving the analysed concerns, described as follows.

'Expanding' Architectural Concerns. A careful analysis of Figure 7-5 (and Appendix B) shows that most of the concerns tend to become more complex as the system evolves. For example, observing the lines of Album and Persistence plotted in Figure 7-5, the number of components forming their crosscutting pattern instances increases as the system evolves. This characteristic of concerns may be associated with later design instability as we investigate in the next section. There also are some particular cases where the number of components forming crosscutting patterns remains almost stable throughout the releases. For instance, Copy and Sorting are representative concerns where the number of components forming the crosscutting patterns remains almost stable, although Copy varies when components are forming a Black Sheep instance.

'Shrinking' Architectural Concerns. An intriguing situation occurs with the Photo concern in the transition of Release 5 to Release 6 where the number of components in crosscutting patterns reduced. For example, as presented in Figure 7-5, 11 Photo-related components are forming an Octopus instance in Release 5, but only 9 in Releases 6 and 7. After investigating the crosscutting pattern instances related to Photo, we found that this concern was deeply affected by the introduction of two new types of media: Music (Release 6) and Video (Release 7). For instance, many architectural components (e.g., `MediaController`[8] in Figure 7-4) which were

[8] MediaController used to be called PhotoController until Release 5.

exclusively dedicated to implement Photo in Release 5 became either part of the Media concern or shared between Photo and Music in Releases 6.

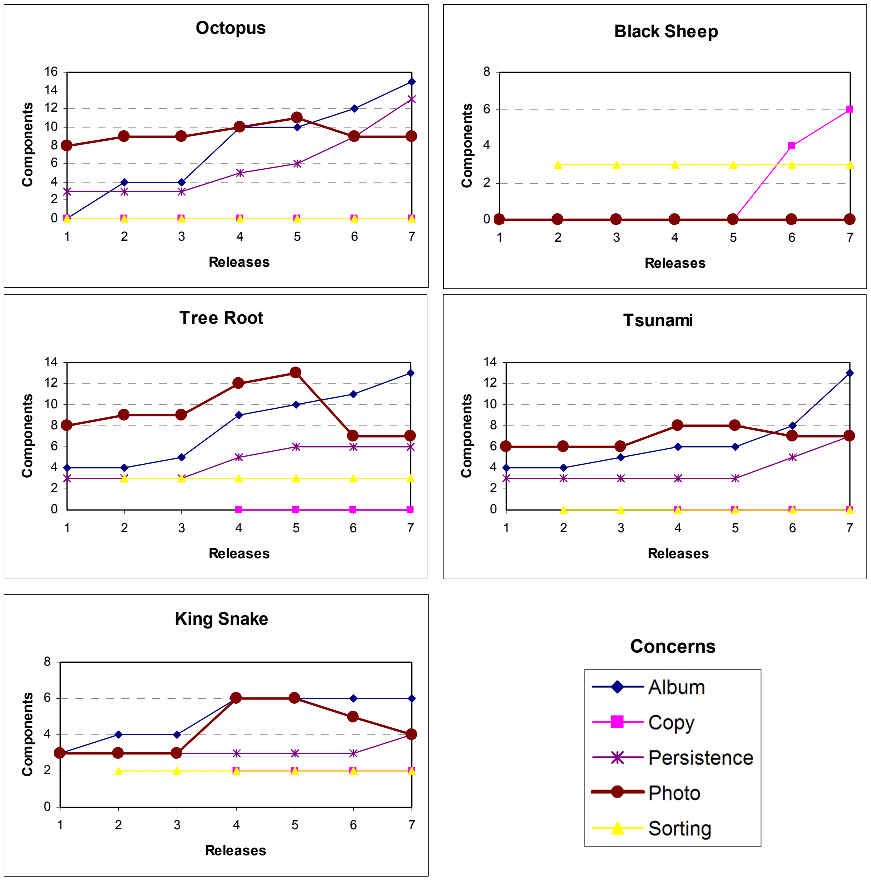


Figure 7-5: Components forming each crosscutting pattern in all releases

7.5 Analysis of Architecture Stability

After we have identified the number of crosscutting patterns in each release, this section focuses on the analysis of architecture stability. Our aim is to assess the ability of crosscutting patterns to predict instabilities of an evolving architecture based on the MobileMedia study results. More specifically, we aim to address the following research questions derived from Hypothesis 7-1 presented earlier in this chapter.

RQ 1. Can the number of crosscutting patterns help to anticipate sources of architecture instability?

RQ 2. Are certain types of crosscutting patterns more correlated with architecture instabilities than others?

To try to answer these questions, we followed three steps detailed in the next subsections. Section 7.5.1 presents the design stability measurement of the MobileMedia architecture. Architecture stability was quantified based on the number of changes each architectural component undertook during the system evolution. After these measures have been calculated, we verify in Section 7.5.2 any correlation between the number of changes and the number of crosscutting patterns in a component. Finally, Section 7.5.3 reports similar analysis considering specific crosscutting patterns and their correlation with changes.

7.5.1 Quantifying Component Changes

Traditional change impact metrics [84, 147] allow us to quantify the propagation of actual architecturally-relevant changes. We calculated the total number of additions

Table 7-3: List of changes in architectural components of MobileMedia.

Components	MobileMedia Releases								# of Changes	Stable
	00	01	02	03	04	05	06	07		
AddMediaToAlbum	a	1	0	0	1	1	1	1	5	no
AlbumController					a	0	1	0	1	yes
AlbumListScreen	a	0	0	0	1	0	1	0	2	yes
AlbumMusicData							a	0	0	yes
AlbumPhotoData	a	1	1	0	1	0	1	0	4	no
AlbumVideoData								a	0	yes
BaseController	a	1	1	1	1	0	1	0	5	no
CaptureMediaContr.								a	0	yes
CaptureMediaScreen								a	0	yes
ImageAccessor	a	1	0	0	1	0	1	0	3	no
MediaController			a	1	1	0	1	1	4	no
MediaListController					a	0	1	0	1	yes
MediaListScreen	a	0	1	1	0	0	1	1	4	no
MusicAccessor							a	0	0	yes
NetworkScreen						a	0	0	0	yes
NewLabelScreen	a	0	1	0	1	0	1	0	3	no
PhotoViewController					a	1	1	1	3	no
PhotoViewScreen	a	0	0	0	1	1	0	1	3	no
PlayMusicController							a	0	0	yes
PlayMusicScreen							a	0	0	yes
PlayVideoController								a	0	yes
PlayVideoScreen								a	0	yes
SMSController						a	1	1	2	yes
VideoAccessor								a	0	yes

and changes in each component per release of the MobileMedia architecture. The use of these metrics supports the identification of the unstable architectural components of MobileMedia; i.e., components that changed many times during the architecture evolution. In other words, we computed the number of times an element changed across the releases.

Table 7-3 presents the number of changes for all components of the MobileMedia architecture. This table also indicates the release that a component changed (intermediary columns). The label 'a' indicates a component added in that specific release. The last column states whether a components is stable ('yes') or unstable ('no'). Components changing in more than 3 releases (i.e., higher than 33% – a typical threshold for ripple effects [147]) were classified as unstable. The two sets of stable or unstable components were then used as the oracle for analysing the predictive power of crosscutting patterns in the rest of this section.

7.5.2 Crosscutting Pattern Density and Architecture Changes

This section presents the first evaluation on the effectiveness of crosscutting patterns to predict architecture instabilities. As discussed earlier in this chapter, we first projected concerns into early artefacts (Section 7.3) and, then, we identified all instances of crosscutting patterns (Section 7.4). We also analysed architectural changes in order to define stable and unstable components of the MobileMedia architecture (Section 7.5.1). Now, we focus our evaluation on the previously stated research question RQ1: "can the number of crosscutting patterns help to anticipate sources of architecture instability?"

In order to answer this question, we analysed the unstable components of the MobileMedia architecture. We also identified two sets of components: (i) components which are taking part in many crosscutting patterns and (ii) components which are not taking part in many crosscutting patterns. Finally, we conducted a Spearman's Rank correlation test [155] between the numbers of crosscutting pattern instances and the list of actual unstable components (Section 7.5.1). In order to evaluate the strength of correlation, we interpreted results based on the following criteria [155]: less than 10% means trivial, 10% to 30% means minor, 30% to 50% means moderate, 50% to 70% means large, 70% to 90% means very large, and more than 90% means almost perfect.

Tables 7-4 and 7-5 illustrate our results in the MobileMedia architecture by showing the components of Release 2 and 6, respectively. Table 7-5 presents only partial results by showing the top-five components (holding more crosscutting patterns) and the bottom-five components (holding fewer crosscutting patterns). The complete data set can be found in Appendix D.

Table 7-4: Unstable architectural components and crosscutting patterns (Release 2).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Changes
	BS	Oct	KS	Tsu	TR		
BaseController	1	3	7	5	6	22	4
AlbumData	0	2	3	5	4	14	3
ImageAcessor	0	1	3	3	4	11	2
PhotoController	1	1	3	1	3	9	4
PhotoListScreen	1	2	1	2	3	9	4
AlbumListScreen	0	2	2	2	2	8	2
NewLabelScreen	0	3	1	2	2	8	3
AddPhotoToAlbumScreen	0	2	1	1	1	5	4
PhotoViewScreen	0	1	0	0	1	2	3
Average	**0.3**	**1.9**	**2.3**	**2.3**	**2. 9**	**9.8**	**3.2**
Correlation	**70%**	**23%**	**17%**	**-6%**	**11%**	**17%**	

Table 7-5: Unstable architectural components and crosscutting patterns (Release 6).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Changes
	BS	Oct	KS	Tsu	TR		
AlbumController	0	6	4	3	7	20	1
AlbumPhotoData	0	5	6	6	6	23	1
ImageAccessor	0	4	5	5	5	19	1
MediaController	2	7	8	7	10	34	2
PhotoViewController	2	7	4	3	9	25	2
<Complete list is in Appendix D>							
AddMediaScreen	0	4	0	2	1	7	2
AlbumListScreen	0	2	2	2	2	8	1
NetworkScreen	0	1	1	1	1	4	0
NewLabelScreen	0	2	2	1	2	7	1
PlayMusicScreen	0	2	1	1	1	5	0
Average	**0.4**	**4**	**3.3**	**3.1**	**4.4**	**15.2**	**1.0**
Correlation	**42%**	**40%**	**44%**	**4%**	**64%**	**51%**	

We choose Release 2 as a representative earlier release, i.e., many change requests were addressed after this release. Additionally, we choose Release 6 to represent a later release. In fact, many modifications were made from Release 5 to Release 6 of MobileMedia in order to accommodate a new type of media (Music) in the system. Therefore, Release 6 presented some interesting situations worth observing. Columns two to six of Tables 7-4 and 7-5 specify the number of instances for each architectural crosscutting pattern: Black Sheep (BS), Octopus (Oct), King Snake (KS), Tsunami (Tsu), and Tree Root (TR). In addition to the number of crosscutting patterns, these tables present the number of changes in the respective architectural components (last column). We only took into consideration changes that occurred after each analysed release.

High Incidence of Crosscutting Patterns Correlates with Component Instabilities in Later Architecture Releases. An interesting situation can be recognised in Table 7-5: components which usually changed are also the locus of more crosscutting pattern instances (top components). For example, components with about 20 pattern instances changed at least once after Release 6 (inclusive). On the other hand, two components with less than 5 crosscutting patterns have not changed in Releases 6 and 7. These results are supported by Spearman's correlation which indicates a large correlation degree (51%) between the number of crosscutting patterns and module changes. However, the same conclusion can not be drawn in the analysis of Release 2 (Table 7-4). Spearman's correlation indicates just a minor correlation degree (17%) in this release. Hence, our analysis based on the overall number of crosscutting patterns may serve as an indicator of architecture instability in some situations, such as later architecture release.

7.5.3 Specific Crosscutting Patterns and Architecture Changes

The results of our analysis also allow us to verify whether or not a specific crosscutting pattern is a good indicator of architecture instability. That is, we can try to answer the second research question (RQ2) of this section: "Are certain types of crosscutting patterns more correlated with architecture instabilities than others?" To support this analysis, we rely on the Spearman's Rank correlation test [155] between the number of each crosscutting pattern and the number of change in a component presented in the last line of Tables 7-4 and 7-5.

This correlation test suggests that none of the crosscutting patterns are strongly correlated to architecture stability in both analysed releases, although all of them (except Tsunami) presented minor to moderate degrees of correlation in at least one release. The best correlation is achieved by Black Sheep in Release 2 (70%).

However, only three architectural components (BaseController, PhotoController, and PhotoListScreen) presented this crosscutting pattern what limits our conclusions. Moreover, the correlation of Black Sheep instances and changes was just moderate (42%) in Release 6. For three crosscutting patterns, Octopus, King Snake, and Tree Root, there are always a minor correlation degree in Release 2 and moderate one in Release 6. Our data for Tsunami do not allow us to draw any conclusion about the correlation of this specific crosscutting pattern and architecture stability. Chapter 8 will report that some of these crosscutting patterns are better indicators of instabilities at the design level.

7.6 Threats to Validity

This section discusses some threats to the validity of the study presented in this chapter. We should emphasise that our findings in this study are restricted to the target system, its architecture models, and the analysed concerns. In other words, results regarding the possible correlations between crosscutting patterns and architecture stability may not be directly generalised to other contexts. However, this first exploratory study allowed us to draw useful conclusions of whether specific crosscutting patterns can be used to indicate sources of architecture instabilities. In addition, this study also allowed us to make useful evaluation about the applicability and usefulness of the heuristic technique for detecting crosscutting patterns.

The set of selected concerns includes 14 representative concerns of 3 different categories, namely features, non-functional requirements, and roles of architectural patterns. Similarly, we also rely on a set of 8 change scenarios applied to the architecture models of the target system. The change scenarios target at the inclusion not only of mandatory and varying features but also of non-functional requirements. However, we acknowledge that the sets of concerns and change scenarios may not be large enough to enable broad conclusions. In fact, it is clear for us that the number of concerns and change scenarios used in the study is by no means statistically relevant. Nonetheless, we are considering the sample representative of the population due to the heterogeneity of concerns and change scenarios involved in this study. In addition, we presented the complete data set and we tried to explain the results based on qualitative analysis. Moreover, we described the assessment methodology we used which enables further replications of our exploratory study.

The conclusions are particularly limited by the projection of concerns into architecture models which is required for the identification of crosscutting patterns. Variations in the concern projection could lead to different crosscutting patterns identified. As mentioned in Section 7.2.1, we followed some guidelines to make this process more reliable. For instance, we relied on pair concern projection. That is, the projection of each concern into architecture models was performed by two people assisting each other. We also assigned researchers with previous knowledge of specific concerns to project them (Section 7.2.1). Moreover, as we observed in our controlled experiment with students (Section 6.2), even developers who do not have much experience in projecting concerns are able to achieve good accuracy in this task.

Another issue worth to mention is the selection of threshold values in this study. This problem is not new and it characterises most of the quantitative approaches. Setting the threshold values is a highly empirical process and it is usually guided by similar past experiences and by hints from previous studies [103]. In our particular case, we defined threshold values, for instance, to decide when a component is stable or unstable (Section 7.5.1). We used the number of changes greater than 33% of the releases to define unstable components. In fact, we have consistently used threshold of 33%, 50%, and 67% in this document. These values are in accordance with thresholds used by other authors in quantitative studies [95, 103].

7.7 Summary

The key goal of this chapter was to investigate the hypothesis that some specific crosscutting patterns are more harmful to design stability than others. To investigate this hypothesis, this chapter reported the results of an empirical study on the stability assessment of software architecture based on crosscutting patterns. This study relied on the projection of 14 concerns in the architecture models of a software product line (Section 7.1). Based on the results of this study, we confirmed that concerns tend to manifest more complex crosscutting patterns as the system evolves which may be related to the deterioration of its architecture stability. In some particular situations, however, the introduction of a concern reduces the overall complexity of other concerns (Section 7.4).

We also observed that, in later releases, the high incidence of crosscutting patterns may correlate with component instabilities even in early development artefact, such as architecture models (Section 7.5.2). Moreover, by analysing particular crosscutting patterns, we found that none of the investigated crosscutting patterns are strongly correlated to architecture stability. Some crosscutting patterns, such as Octopus and King Snake, presented moderate correlation in later architecture releases. However, there is not evidence of correlation for other crosscutting patterns, such Tsunami, in the analysed architecture models (Section 7.5.3). Since our results are restricted to the selected concerns and models, further analyses may be required to confirm or refute our findings. The next chapter replicates this study relying on detailed design models and implementation artefacts. That is, it assess whether crosscutting patterns can be used as stability indicators in low-level software artefacts.

8. Assessment of Design Stability

The high occurrence of crosscutting concerns in a component is likely to be associated with different maintainability properties of the system, such as design stability. Supporting this belief, a recent work from Marc Eaddy and his colleagues [44] has shown that the more scattered a concern is the more faults it causes to components realising it. Faults in a component may lead to its instability because changes are required to fix these faults. However, as discussed in Section 2.5.2 concern scattering and tangling by themselves are not good indicators of design stability. For instance, the `MetaObserver` and `MetaSubject` interfaces might be sources of instability in the design of Figure 2-5, although these interfaces realise a single concern. That is, changing a method signature of these interfaces, for instance, would trigger many other updates to elements realising the Observer design pattern. Therefore, assessment of design stability should consider several properties of crosscutting concerns as documented by the crosscutting patterns in Chapter 4.

In this context, this chapter reports an empirical study to investigate the impact of crosscutting patterns on design stability. We use our tool-supported heuristic technique to detect instances of crosscutting patterns in three heterogeneous applications (Section 8.1). The first one, MobileMedia, was also used in the study presented in the previous chapter. This previous study relied on architecture models of MobileMedia to empirically evaluate the impact of crosscutting patterns on architecture stability. The other two applications, Health Watcher and a Design Pattern library, were selected from our analysis of crosscutting concern detection (Section 6.3). Based on the analysis of concerns in the designs of these systems, this chapter aims to investigate the following hypothesis.

> ***Hypothesis 8-1****: Some specific crosscutting patterns are more harmful to design stability than others.*

To test this hypothesis, our evaluation relies on a set of 67 concern instances from the three target applications. This set includes the 14 concerns investigated in the previous chapter which are now analysed at the detailed design level. Moreover, we also consider additional design patterns and functional and non-functional requirements. We analysed two different decompositions of the target systems, an object-oriented one and an aspect-oriented one. These concerns are projected into both object-oriented and aspect-oriented decompositions. In addition, several successive releases of two of the chosen systems are used.

The research methodology adopted in this chapter is similar to the one described in the previous chapter (Section 8.2). That is, it consists of detecting the crosscutting patterns in the design artefacts of the target systems. After detecting all crosscutting patterns, we quantify the number of crosscutting pattern instances that can be found in each component, the *crosscutting pattern density* (Section 8.3). Similarly, the number of changes in a design component is also calculated in order to compare this instability symptom with the crosscutting pattern density. Section 8.4 performs such analysis and tries to correlate crosscutting patterns and design stability. Section 8.5 discusses constraints in the study validity and Section 8.6 summarises this chapter.

8.1 Selection of the Target Applications

The evaluation step performed in this chapter uses three applications selected from previous chapters. The first application, called MobileMedia (Section 8.1.1), is a software product line for handling data on mobile devices. Its architectural models were used in the evaluation performed in Chapter 7. The other two systems, namely Health Watcher (Section 8.1.2) and a Design Pattern library (Section 8.1.3) were selected from the six applications used in our preliminary evaluation of crosscutting concern identification presented in Section 6.3. For all three applications, Java and AspectJ implementations were previously available [56, 73, 80, 148]. The analysis of crosscutting patterns in these systems is particularly interesting because they come from significantly different application domains. These three applications were selected because they met relevant criteria for our evaluation process as discussed in the next subsections.

8.1.1 The MobileMedia Design

The architecture design of MobileMedia using the Component & Connector notation was presented and discussed in Section 7.1. Unlike the analysis we presented in the previous chapter, the study of this chapter relies on information gathered from class diagrams, such as inheritance relationships, which is not available in the MobileMedia architecture models. Figure 8-1 presents a partial UML class diagram derived from the previously presented Component & Connector model of MobileMedia. The MobileMedia design is mainly determined by the use of the Model-View-Controller (MVC) architectural pattern [20]. Components realising each of the three roles of the MVC pattern are indicated in Figure 8-1 by the three grey boxes. Object-oriented and the aspect-oriented versions of MobileMedia have been implemented in Java and AspectJ, respectively.

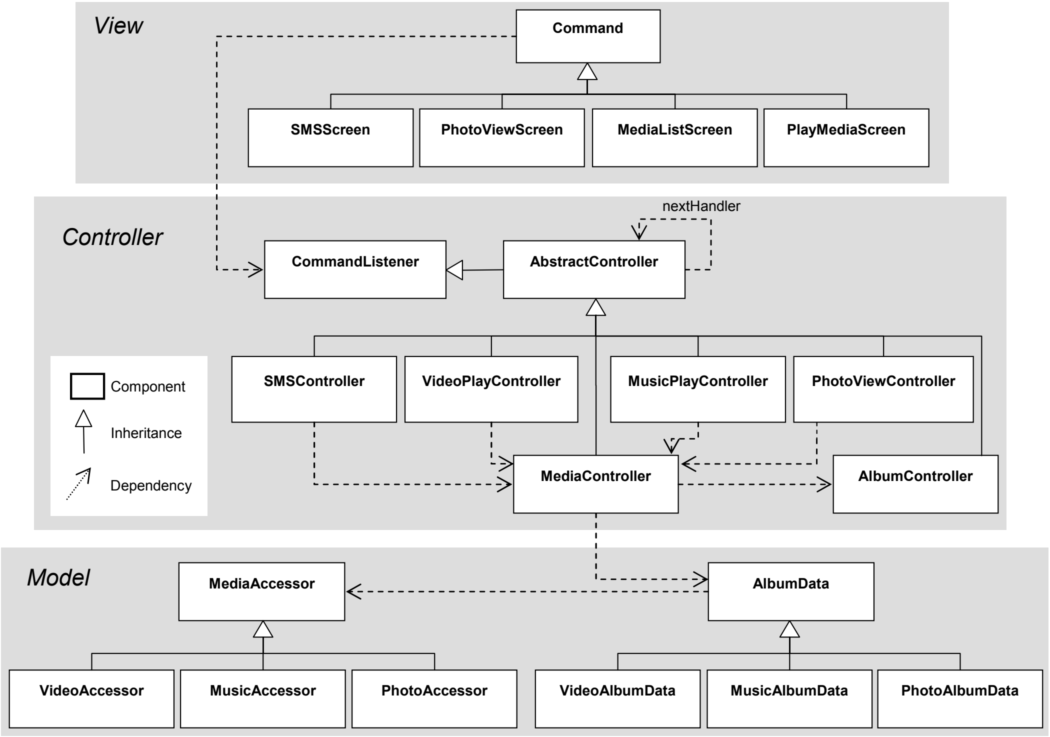


Figure 8-1: Partial design of MobileMedia

8.1.2 The Health Watcher System

The second target application is a medium-sized Web-based information system, called Health Watcher [73, 134]. Health Watcher allows a citizen to register complaints to the public health system. Using a UML diagram [121], Figure 8-2 shows a simplified design model representing some classes and interfaces of the system. In Health Watcher, complaints are registered, updated, and queried through a Web client. The system is structured in layers, using the Layers architectural pattern [20], with the goal of decoupling different parts of the system and making these parts easy to change independently.

In Health Watcher, complaints are managed by means of Java Servlets [82], represented by the components in the view layer. Accesses to the Health Watcher services are made through the `IFacade` interface, which is implemented by the `HealthWatcherFacade` class. This class works as a portal to access the business collections, such as `ComplaintRecord` and `EmployeeRecord`. Records access the data layer using interfaces, like `IComplaintRepository`, which decouple the business logic from the specific type of data management. For instance, Figure 8-2 shows the `ComplaintRepositoryRDB` component that implements a repository for a relational database.

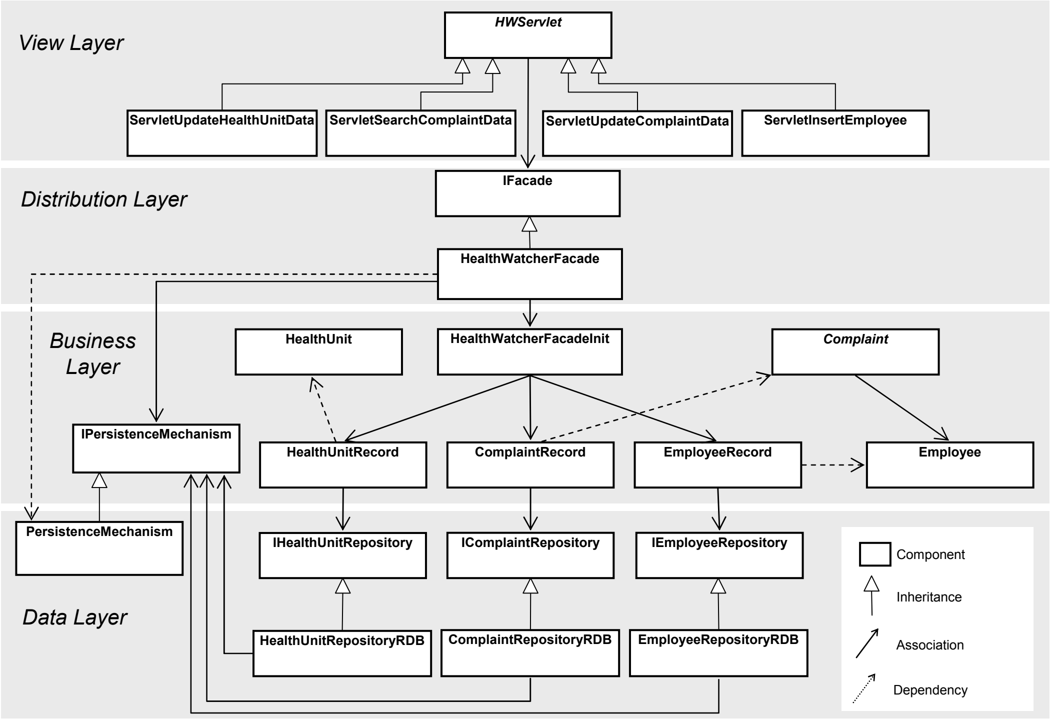


Figure 8-2: Partial design of Health Watcher

This structure prevents some code scattering and tangling because it clearly separates some of the Health Watcher main concerns in layers. However, tangling is not completely avoided [137]. For example, `HealthWatcherFacade` implements several concerns, including Persistence (transaction management) and Distribution. One possibility is the use of adapters [64] to take care of transaction functionality, but at a high price, since developers must maintain both the facade and the adapter. Similarly, the Health Watcher design also fails to completely prevent code scattering. For instance, almost every component must contribute to the realisation of the Exception

Handling concern. However, despite not completely separating concerns, the layered design gives some support for changes. For instance, in the system configuration one could use Enterprise Java Beans (EJB) instead of RMI.

Table 8-1 summarises the main responsibilities associated with each group of classes in the Health Watcher system. For instance, the first group includes classes, such as `HWServlet` and `ServletInsertEmployee`, which implement the user interface (Web) of the system.

Table 8-1: Key responsibilities of the Health Watcher classes

Classes	What they do
HWServlet ServletInsertEmployee, ServletSearchComplaintData, ServletUpdateComplaintData, ServletUpdateHealthUnitData	These classes implement the user interface of the system.
IFacade, HealthWatcherFacade, HealthWatcherFacadeInit	These classes provide a simple interface to all services of the system.
HealthUnit, Employee, Complaint	These classes represent business basic concepts of the Health Watcher system domain, such as health units, employees, and complaints.
HealthUnitRecord, EmployeeRecord, ComplainRecord	These classes represent groups of objects for the corresponding basic classes. For instance, EmployeeRecord groups objects of Employee.
HealthUnitRepositoryRDB, EmployeeRepositoryRDB, ComplainRepositoryRDB	These classes contain methods for manipulating persistent objects of the basic classes. The code of these classes depends on a specific API for accessing the persistence platform, thus any changes to this platform will directly impact these classes.
IHealthUnitRepository, IEmployeeRepository, IComplainRepository	These interfaces establish a decoupled relationship between the “Record” classes and “RepositoryRDB” classes. In this way, changes to the data access code (i.e., “RepositoryRDB” classes) do not have impact on business code.
PersistenceMechanism, IPersistenceMechanism	This class and its interface implement services such as connecting to and disconnecting from the database, transaction mechanism, concurrency mechanism, etc.

Health Watcher was selected because it met a number of relevant criteria for our study. First, it is part of a real-world health care system used by the city of Recife in Brazil. Also, it has existing Java and AspectJ implementations with around 7 KLOC each. The Health Watcher system complements MobileMedia (our first application) with several kinds of crosscutting and non-crosscutting concerns present in typical Web-based information systems. Health Watcher also involves a number of recurring concerns and technologies common in day-to-day software development, such as GUI, Persistence, Concurrency, RMI, Servlets, and JDBC. Moreover, the Health Watcher design and implementation choices have been extensively discussed [73, 137] and both object-oriented and aspect-oriented solutions evolved in a controlled manner. As MobileMedia, the Health Watcher system was also developed with modularity and maintainability principles in mind. Finally, the first Health Watcher release of the Java implementation was deployed in March 2001, since then a number of incremental and perfective changes have been addressed in posterior Health Watcher releases. These changes allow us to observe typical types of changes in this application domain.

8.1.3 A Library of Design Patterns Implementations

Several design patterns exhibit crosscutting structures [66, 80]. Based on this, Hannemann and Kiczales undertook a study [80] in which they developed a library of Java [38] and AspectJ [94] implementations of the 23 Gang-of-Four design patterns [64]. This library of implemented design patterns is the third case study of this chapter. Hannemann and Kiczales claim that programming languages affect pattern implementations. Hence, it is natural to explore the effect of aspect-oriented programming techniques on the implementation of design patterns. For each of the 23 Gang-of-Four design patterns, they developed a representative example that makes use of the design pattern. Then, they implemented the example in both Java and AspectJ.

To illustrate one design pattern implementation of this library, Figure 8-3 depicts the class diagram of the object-oriented implementation of the Mediator pattern. The intent of this design pattern is to define an object that encapsulates how a set of objects should interact [64]. The Mediator pattern defines two roles – Mediator and Colleague – to its participant classes. The Mediator role has the responsibility for controlling and coordinating the interactions of a group of objects. The Colleague role identifies objects that need to communicate with each other. Hannemann and Kiczales [80] present a simple example of the Mediator design pattern in the context of a Java

Swing application (Figure 8-3). In this example, the Mediator pattern is used to manage the communication between two kinds of graphical user interfaces components. The `Label` class plays the Mediator role of the design pattern and the `Button` class plays the Colleague role.

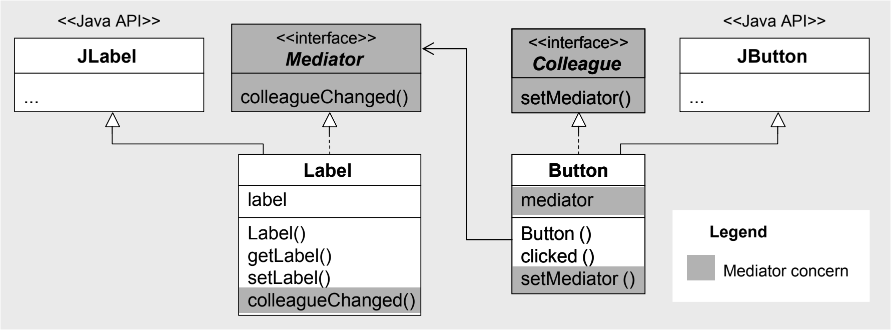


Figure 8-3: Object-oriented design of the Mediator pattern

In Figure 8-3, the `Mediator` and `Colleague` interfaces define the roles of the Mediator design pattern. Specific application classes must implement these interfaces based on the role that they need to play. In the example presented, the `Button` class implements the `Colleague` interface. The `Label` class implements the `Mediator` interface in order to handle actions when buttons are clicked. Figure 8-3 also illustrates how the object-oriented implementation of the Mediator pattern is spread across the code of the application classes. The shadowed grey attributes and methods represent code required to implement the Mediator design pattern in the application context.

We selected this library as a third target study because design patterns are reusable solutions which appear across a large range of applications. Therefore, understanding the crosscutting nature of design patterns is a matter of the utmost importance. Furthermore, design patterns generally assign roles to their participants which may crosscut the class core functionality. This characteristic is an extra motivation for our analysis based on crosscutting patterns. For instance, the Mediator and Colleague roles defined in the Mediator design pattern crosscut the base functionality of the `Label` and `Button` classes. Hence, some operations that change the `Button` state (Colleague) must trigger updates to the corresponding Mediator. In other words, the act of updating crosscuts one or more operations in each Colleague class in the Mediator design pattern. The crosscutting structures of the Gang-of-Four design patterns have also been detected and explored in previous investigations [22, 67, 80]. However, their specific crosscutting patterns have not yet documented.

8.2 Experiment Setting Outline

This section describes the configuration of this empirical study. When setting this study, we first selected a set of concerns of each system (Section 8.2.1). We then selected a set of change scenarios (Section 8.2.2) for two of the chosen systems, namely MobileMedia and Health Watcher. Finally, we performed a set of experimental tasks to analyse the impact of crosscutting patterns on design stability (Section 8.2.3).

8.2.1 Selection of Concerns

Two different situations occurred in the selection of concerns for each system. First, six researchers worked together to identify and project 14 concerns in the architecture and design artefacts of MobileMedia as described in Section 7.2.1. We also analysed 11 and 44 concerns of Health Watcher and the Design Pattern library, respectively, presented in this section. These concerns and their projections were available from previous studies [66, 73, 80].

Health Watcher. The analysed concerns in Health Watcher include four typical crosscutting concerns: Concurrency, Distribution, Persistence, and Exception Handling. According to previous studies [73, 137], these concerns were selected because they are the main modularisation target either at the initial Health Watcher design decomposition (Section 8.1.2) or at the change scenarios (Section 8.2.2). For the same reasons, two layers were also analysed: View and Business. Moreover, the concerns relative of five key design patterns (Abstract Factory, Adapter, Command, Observer, State) adopted during the system evolution were considered. Table 6-1 (Section 6.2) described the analysed concerns of Health Watcher, except the design patterns. The design patterns have the same definitions given by the Gang-of-Four book [64]. These definitions are also used in our third study.

The Design Pattern library. For the third case study, we analysed 44 roles of the 23 Gang-of-Four design patterns [64] as presented in Table 8-2. For example, the Mediator and Colleague roles were analysed in the implementation of the Mediator design pattern (Figure 8-3), while the Context and State roles were investigated in the implementations of the State design pattern. We took instances of the 23 design patterns implemented by Hannemann and Kiczales [80]. Like their study, we consider each pattern role as a concern because the roles are the primary sources of crosscutting structures. Since the crosscutting structures of these design patterns have already been detected and explored in previous studies [22, 66, 80], we used the available concern projection.

Table 8-2: Roles analysed per design pattern

Design Pattern	Roles
Abstract Factory	Factory and Product
Adapter	Adaptee, Adapter, and Target
Bridge	Abstraction and Implementor
Builder	Builder and Director
Chain of Responsibility	Handler
Command	Command, Invoker, and Receiver
Composite	Component, Leaf, and Composite
Decorator	Component and Decorator
Façade	Façade
Factory Method	Creator and Product
Flyweight	Flyweight and Flyweight Factory

Design Pattern	Roles
Interpreter	Context and Expression
Iterator	Aggregate and Iterator
Mediator	Mediator and Colleague
Memento	Memento and Originator
Observer	Observer and Subject
Prototype	Prototype
Proxy	Proxy
Singleton	Singleton
State	Context and State
Strategy	Context and Strategy
Template Method	Abstract Class and Concrete Class
Visitor	Element and Visitor

8.2.2 *Change Scenarios*

We considered 8 successive releases of MobileMedia [56, 116] and 10 releases of Health Watcher [73]. The MobileMedia releases were described in Table 7-2 (Section 7.2). Each release of Health Watcher was changed by adding new functionalities to its specific application or by refactoring its design to achieve a better modularised structure. We only analysed one release of the Design Pattern library. Although the design patterns have not been changed, they supported the analysis of concerns recurrently found in evolving software systems.

Health Watcher. The selected changes of the Health Watcher system vary in terms of the types of modifications. While some of them add new functionality, some others improve or replace functionality, and others improve the system structure for better reuse or modularity. Their main purpose is to expose the object-oriented and aspect-oriented designs to distinct maintenance tasks that are recurring in incremental software development. These changes originate from many sources. For instance, the original developers of Health Watcher implemented changes to meet new stakeholders' requests (that are actually necessary) or just to improve the system structure (that are not compulsory). There are also changes created by researchers involved in previous studies [73, 137], where certain extensions and improvements were implemented. Before the latter changes were applied, the original developers of Health Watcher had been consulted to confirm whether these changes were valid. This variety of sources of changes aims to not artificially bias the results of empirical studies. Table 8-3 summarises the set of changes considered in this study for this application.

Table 8-3: Summary of scenarios in Health Watcher

	Change Scenarios	Impact
0	The base Health Watcher release	-
1	Factor out multiple Servlets to improve extensibility.	View Layer
2	Ensure the complaint state cannot be updated once closed to protect complaints from multiple updates.	View/Business Layers
3	Encapsulate update operations to improve maintainability using common software engineering practices.	Business/View Layers
4	Improve the encapsulation of the distribution concern for better reuse and customisation.	View/Distribution/ Business Layers
5	Generalise the persistence mechanism to improve reuse and extensibility.	Business/Data Layers
6	Remove dependencies on Servlet response and request objects to ease the process of adding new GUIs.	View Layer
7	Generalise distribution mechanism to improve reuse and extensibility.	Business/View/ Distribution Layers
8	New functionality added to support querying of more data types.	Business/Data/View Layers
9	Modularise exception handling and include more effective error recovery behaviour into handlers.	Business/Data/View Layers

8.2.3 Study Outline

Unlike the study presented in the previous chapter, we did not need to measure concerns in this evaluation step. All crosscutting patterns were automatically identified in implementation artefacts by our tool (Section 5.6) – which also quantifies the concerns behind the scenes. Although we used the tool in implementation artefacts, we understand that our results can be generalised to the design level. That is, the required design information can be easily extracted from the system implementations. This evaluation step is divided in three parts described as follows.

1. **Measurement of Crosscutting Pattern Density**. After all concerns have been classified according to their respective crosscutting patterns, we quantified the number of crosscutting pattern instances per component (Section 8.3).

2. **Measurement of Component Stability**. Similarly to the previous step, we also quantified the number of changes per component. This step allowed us to identify the most unstable components of each target application. Note that, this step was independently performed at both architecture and design models. Hence, the unstable components identified in the architecture models might not be the same unstable components identified at the design level.

3. **Correlation of Crosscutting Patterns and Component Stability**. Finally, we compared the components with high number of crosscutting patterns and with high incidence of changes (Section 8.4). This step aimed to show which kinds of crosscutting patterns are good indicators of design changes (and which ones are not).

We performed Steps 1 to 3 for MobileMedia and Health Watcher. However, we only performed Step 1 for the Design Pattern library since we only analysed one release of each design pattern implementation. The remainder of this chapter discusses the results of our evaluation following the three steps described above.

8.3 Crosscutting Pattern Density in Design Models

This section reports the number of crosscutting pattern instances detected by our tool in the target applications. Section 8.3.1 presents the crosscutting patterns identified in the object-oriented designs. Section 8.3.2 presents the crosscutting patterns detected aspect-oriented designs.

8.3.1 Density in Object-Oriented Designs

This section reports and discusses the patterns of crosscutting concerns detected in the object-oriented designs of the three analysed applications (Section 8.1). Table 8-4 presents partial data regarding the number of crosscutting pattern instances in each object-oriented design (and multiple releases in the MobileMedia case). We take into consideration all selected concerns of these systems (Section 8.2.1). Rows of this table list the 13 crosscutting patterns while columns show the systems. The last row indicates the number of concerns analysed in that specific application. The number of concerns varies through the MobileMedia releases because new concerns were introduced as the system evolved [56]. We focus just on the key findings, but complete results for all data per concern are available in Appendix C and at a supplementary website [35].

Table 8-4: Crosscutting patterns in object-oriented designs

Crosscutting Patterns	**MobileMedia Releases**								**HW R9**	**DPL**
	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**		
Black Sheep	0	1	1	2	2	2	2	2	1	5
Octopus	2	3	5	5	6	7	8	10	5	11
God Concern	1	1	1	1	1	1	0	0	2	10
Climbing Plant	3	4	4	4	7	8	9	10	6	25
Hereditary Disease	0	0	0	0	0	0	7	9	2	9
Tree Root	3	3	4	4	7	6	7	9	8	6
Tsunami	3	3	4	4	5	6	8	9	6	16
King Snake	3	3	3	4	8	7	8	11	4	8
Neural Network	3	3	3	4	8	7	8	11	4	4
Copy Cat	0	2	2	2	2	2	3	3	6	11
Dolly Sheep	0	0	0	0	0	0	0	0	0	2
Data Concern	0	0	0	0	0	0	0	0	0	1
Beh. Concern	0	0	0	0	0	0	0	0	0	10
Num. of concerns	**5**	**5**	**6**	**7**	**8**	**9**	**11**	**13**	**11**	**44**

Table 8-4 shows that some crosscutting patterns are more common in some particular cases. For example, the last three patterns (Dolly Sheep, Data Concern, and Behavioural Concern) could only be identified in the Design Pattern library (the DPL column). This result is somehow expected due to the interesting heterogeneous forms of concerns found in design patterns. Someone could alternatively argue that, based on our results, such crosscutting patterns seem to be specific to design pattern implementations. However, the Gang-of-Four design patterns and their variants [64] are consistently instantiated in all types of application domains and, therefore, should be carefully analysed.

Furthermore, Hereditary Disease instances only emerged in situations with frequent use of inheritance, such as Health Watcher (HW), the last two MobileMedia releases, and some design patterns. This type of crosscutting patterns seems to be particularly interesting to analyse as they often manifest in program families, such as frameworks and product lines [32, 124]. Program families rely extensively on the use of abstract classes and interfaces in order to implement variabilities. The inappropriate modularisation of crosscutting concerns in this category might lead to future instabilities in the design of the varying components. There also are six crosscutting patterns (Octopus, Climbing Plant, Tree Root, Tsunami, King Snake, and Neural Network) which consistently appeared in all applications. On the other hand, three other patterns (Black Sheep, God Concern, and Copy Cat) appeared in small numbers, although uniformly throughout the case studies.

8.3.2 Density in Aspect-Oriented Designs

We also applied our heuristic technique to detect crosscutting patterns in the corresponding aspect-oriented designs of the three target systems. Table 8-5 shows the number of crosscutting pattern instances found in some aspect-oriented designs of our studies. Complete data are presented in Appendix C. Matching our intuition, the aspect-oriented solutions revealed fewer crosscutting patterns since they tend to better separate crosscutting concerns. The overall reduction of crosscutting patterns was around 61% (486 in object-oriented against 298 in aspect-oriented designs). In particular, aspect-oriented programming mechanisms were responsible for the removal of almost 80% of the Black Sheep and Copy Cat instances. That is, just 4 instances of Black Sheep and 7 instances of Copy Cat remained in all aspect-oriented designs (against 18 and 33 in object-oriented designs, respectively). This observation suggests that aspect-oriented programming mechanisms easily address tiny scattered concerns and replicated code; for instance, by means of quantifiers and wildcards.

Table 8-5: Crosscutting patterns in aspect-oriented designs

Crosscutting Patterns	MobileMedia Releases								HW R9	DPL
	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**		
Black Sheep	0	1	0	0	0	0	0	0	0	3
Octopus	2	3	5	5	5	5	5	5	0	8
God Concern	1	1	1	1	1	1	0	0	0	2
Climbing Plant	3	3	3	3	4	5	4	7	2	5
Hereditary Disease	0	0	0	0	0	0	6	8	0	0
Tree Root	3	3	4	4	4	7	8	8	3	7
Tsunami	3	3	3	3	3	5	6	7	2	6
King Snake	3	3	4	4	5	6	7	7	0	4
Neural Network	3	2	2	3	7	7	7	9	1	1
Copy Cat	0	0	0	0	0	0	2	2	2	1
Dolly Sheep	0	0	0	0	0	0	0	0	0	1
Data Concern	0	0	0	0	0	0	0	0	0	0
Beh. Concern	0	0	0	0	0	0	0	0	0	5
Num. of concerns	**5**	**5**	**6**	**7**	**8**	**9**	**11**	**13**	**11**	**44**

Tables 8-4 and 8-5 also show that some crosscutting patterns may not be easily addressed by the aspect-oriented programming mechanisms. For instance, only 10% of Tree Root instances were removed by the use of aspects (57 instances in object-oriented against 51 instances in aspect-oriented designs). In fact, we noticed that new Tree Root instances are even created with the use of aspects. This is the case, for instance, when the aspect-oriented programming mechanisms are unable to completely modularise a concern and several concrete aspects depend on an abstract aspect (e.g., protocol aspects in the Design Pattern library). The abstract aspect is the root of a Tree Root instance in this case. In addition to Tree Root, only few instances of other crosscutting patterns in the category of Communicative Concerns (i.e., Tsunami, King Snake, and Neural Network) were removed in the aspect-oriented designs. This result suggests that this category of crosscutting patterns, which involves high coupling between the participating components, is not easily addressed by the aspect-oriented programming adopted in the target applications.

8.4 Analysis of Design Stability

This section reports the results of three steps aiming to analyse design stability of two target applications: MobileMedia and Health Watcher. First, we measure the design stability based on the component changes during the evolution of the target software systems (Section 8.4.1). Then, we try to correlate the tally number of crosscutting patterns with changes in a component (Section 8.4.2). Finally, we verify the correlation of specific crosscutting patterns and design changes (Section 8.4.3). Although we collected data for both object-oriented and aspect-oriented designs, we try to generalise our results without considering technique-specific particularities. Our aim is to verify whether crosscutting patterns correlate with design stability regardless of the used software decomposition technique. Hence, our research questions which support Hypothesis 8-1 of this chapter are similar to the ones presented in Section 7.5. They can be formalised as follows.

RQ 1. Can the number of crosscutting patterns help to anticipate sources of design instability?

RQ 2. Are certain types of crosscutting patterns more correlated with design instabilities than others?

8.4.1 Quantifying Component Changes

This section presents the stability measurements performed in both object-oriented and aspect-oriented designs of the two applications. This phase relies on two typical change impact measures [84, 147] which quantify the number of components (aspects/classes) added and changed in the systems' evolutions. The purpose of using these metrics is to quantitatively assess stability of each design component when applying several changes to the system. In the MobileMedia case, we observed that a change has similar effects for most releases when comparing design changes with the architectural counterparts (Section 7.5.1).

Tables 8-6 and 8-7 summarise the total number of components added and removed per release in MobileMedia and Health Watcher, respectively. From these data, we can verify that there were about 70 component changes in each MobileMedia design (Table 8-6) and 160 component changes in each Health Watcher design (Table 8-7). These sets of changes are used as baseline to verify the correlation of crosscutting patterns and design stability. The detailed data of changes per component of each application are presented in Appendix E.

Table 8-6: Number of changed components per release in MobileMedia

	Design	Releases						
		0-1	1-2	2-3	3-4	4-5	5-6	6-7
Added Components	Object	9	1	0	5	7	10	14
	Aspect	13	2	3	6	8	14	27
Changed Components	Object	5	8	5	8	6	19	13
	Aspect	5	10	2	10	5	26	17

Table 8-7: Number of changed components per release in Health Watcher.

	Design	Releases								
		1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10
Added Components	Object	24	12	2	3	4	4	4	12	5
	Aspect	29	16	3	0	4	4	2	12	6
Changed Components	Object	23	5	14	14	2	26	2	22	48
	Aspect	24	15	14	9	1	28	11	21	51

8.4.2 Crosscutting Pattern Density and Design Changes

This section analyses the correlation between the number of crosscutting patterns and changes in a component. This part of the evaluation comprises three steps. First, we identified the most unstable components by analysing real changes throughout the MobileMedia and Health Watcher software evolutions (Section 8.4.1). Second, we detected each of the 13 crosscutting patterns by analysing all selected concerns in the MobileMedia and Health Watcher designs. Finally, we contrasted the number of crosscutting pattern instances found in each component with the list of unstable components (first step).

Tables 8-8 and 8-9 illustrate our results of MobileMedia by showing 5 of the most unstable components (top components) and 5 of the most stable components (bottom components) for two object-oriented releases (R2 and R6). This analysis focuses on the object-oriented design because crosscutting patterns revealed to be more common in this type of software decomposition technique (Section 8.3). These tables also show the number of changes (last column) and the number of crosscutting patterns (intermediary columns) per component of the respective MobileMedia releases. We took into account only changes which occurred in or after the analysed release. For instance, the last column of Table 8-9 (Total Number of Changes) for Release 6 only considers changes which happened in releases 6 or 7. Considering that this system evolved towards a more stable structure, we focused on Releases 2 and 6 as representative ones of earlier (less stable) and later (more stable) designs. Moreover, these two releases were also analysed in Chapter 7 allowing us to correlate the results gathered at architecture and design levels.

Table 8-8: Unstable MobileMedia components and crosscutting patterns (Release 2)

Components	Crosscutting Patterns (Release 2)							Total of Patterns	Total of Changes
	BS	Oct	HD	TR	Tsu	NN	BC		
PhotoController	1	2	0	2	2	2	1	12	5
ImageUtil	1	4	0	5	1	5	1	22	4
ImageAccessor	0	3	0	5	3	5	0	21	4
BaseController	1	4	0	2	3	3	1	18	4
ImageData	1	2	0	2	4	4	1	17	3
<Complete list is in Appendix E>									
NullAlbumDataR.	0	3	0	0	2	2	0	9	0
U'PhotoAlbumEx.	0	2	0	0	2	2	0	8	0
Pers'MechanismEx.	0	2	0	0	1	2	0	7	0
BaseThread	0	0	0	0	0	0	0	0	0
SplashScreen	0	0	0	0	0	0	0	0	0
Average	**0.2**	**1.9**	**0.0**	**1.4**	**1.4**	**2.1**	**0.2**	**10.3**	**1.5**
Correlation	**69%**	**28%**	**0%**	**43%**	**14%**	**29%**	**67%**	**25%**	

Table 8-9: Unstable MobileMedia components and crosscutting patterns (Release 6)

Components	Crosscutting Patterns (Release 6)							Total of Patterns	Total of Changes
	BS	Oct	HD	TR	Tsu	NN	BC		
AlbumData	0	6	3	6	5	7	0	35	2
MediaController	2	10	0	3	2	3	3	29	2
MediaListControl.	2	7	0	3	2	3	3	25	2
MainUIMidlet	0	3	0	3	5	5	1	22	2
MediaAccessor	0	3	1	4	3	4	0	22	2
<Complete list is in Appendix E>									
NewLabelScreen	0	2	0	0	0	0	0	2	0
SmsSenderThread	0	2	0	0	0	0	0	2	0
BaseThread	0	0	0	0	0	0	0	0	0
Constants	0	0	0	0	0	0	0	0	0
SplashScreen	0	0	0	0	0	0	0	0	0
Average	**0.3**	**3.3**	**0.1**	**1.7**	**1.1**	**2.2**	**0.5**	**12.2**	**0.8**
Correlation	**33%**	**50%**	**28%**	**56%**	**48%**	**62%**	**51%**	**70%**	

Comparing data collected at the design level (Tables 8-8 and 8-9) with data from the MobileMedia architecture (Tables 7-4 and 7-5), we observe that the top unstable components are somehow equivalent. That is, components playing the role of either controllers or data accessors are usually the most unstable ones. Moreover, these components also realise a higher number of crosscutting patterns both at design and architecture levels. With respect to the degrees of correlation between crosscutting patterns and module changes, we observe that these attributes usually correlate better at the design level than they do at the architecture level. For instance, the Spearman's correlation is 17% and 51% at the architecture level for Releases 2 and 6, respectively. On the other hand, at the design level, the correlation is 25% and 70% for the same two releases. Interestingly, both architecture and design analyses based on crosscutting patterns have shown that the correlation between the number of patterns and module changes is higher in later releases, i.e., Release 6 in our case.

Since later releases have presented higher degree of correlation both at design and architecture levels in MobileMedia, we investigate if this situation is confirmed by analysing a later release, Release 9, of Health Watcher. Table 8-10 presents similar information about crosscutting patterns and design stability for the object-oriented design of Health Watcher. We focus our discussion here on these three MobileMedia and Health Watcher releases, but the complete data set is presented in the Appendix E.

Table 8-10: Unstable Health Watcher components and crosscutting patterns (R09)

Components	**Crosscutting Patterns (Release 9)**							**Total of Patterns**	**Total of Changes**
	Oct	**GC**	**HD**	**TR**	**Tsu**	**NN**	**CC**		
Login	3	2	0	3	0	3	2	13	2
UpdateCom'Search	3	2	0	3	0	3	2	13	2
HealthWatcherFac.	4	1	0	3	0	2	2	12	2
ComplaintRe'RDB	3	0	0	3	1	2	2	11	2
GetData'DiseaseType	2	2	0	2	0	2	2	10	2
<Complete list is in Appendix E>									
FacadeFactory	0	0	0	0	0	0	0	0	0
LocalIterator	0	0	0	0	0	0	0	0	0
ServletRequestAd.	0	0	0	0	0	0	0	0	0
ServletResponseA.	0	0	0	0	0	0	0	0	0
Subject	0	0	0	0	0	0	0	0	0
Average	**1.5**	**0.7**	**0.1**	**0.9**	**0.1**	**1.0**	**0.7**	**5.0**	**0.6**
Correlation	**41%**	**57%**	**-10%**	**55%**	**15%**	**53%**	**69%**	**70%**	

Data of Table 8-10 confirm higher correlation of component changes and crosscutting patterns in later releases than in earlier ones. This correlation is the same, 70% which means large correlation, both in Release 6 of MobileMedia and in Release 9 of Health Watcher. Similar to Chapter 7, we rely on the following criteria interpret the strength of correlation [155]: less than 10% means trivial, 10% to 30% means minor, 30% to 50% means moderate, 50% to 70% means large, 70% to 90% means very large, and more than 90% means almost perfect.

It is easy to recognise the large correlation of changes and crosscutting patterns in later releases. For example, components of MobileMedia (Table 8-9) changing twice after Release 6 are affected by at least 20 instances of crosscutting patterns. Analogously, this correlation is also large in Release 9 of Health Watcher. For instance, at least 10 instances of crosscutting patterns were found in the most instable components of Health Watcher (Table 8-10), but none were found in the five most stable components.

8.4.3 Specific Crosscutting Patterns and Design Changes

This section elaborates on the correlation of specific crosscutting patterns and design stability. In order to investigate how each crosscutting pattern relates to design stability, we quantified per component (i) the number of changes and (ii) the number of individual crosscutting patterns. This analysis is based on the results of the Spearman's correlation presented in the last row of Tables 8-8 to 8-10.

Good Correlation with Design Stability. Interestingly, our analysis indicates that all four Communicative Concerns (Section 4.4) present moderate to large correlation with module changes in Release 6 of MobileMedia. This degree of correlation is also confirmed in Release 9 of Health Watcher, except for Tsunami. This finding may explain why modules referencing common blocks of code were found to be the main sources of instabilities in a recent case study [84]. For instance, data of Tables 8-9 and 8-10 show that the correlation of King Snake, Tree Root and Neural Network with module changes is higher than 50% in the later releases of both applications. We also observe from our data that Tsunami does not always follow this large correlation presented by the other patterns in the category of Communicative Concerns (e.g., Table 8-10). This result might be explained by the fact that just few components (the wave source) have actually high coupling in the Tsunami crosscutting pattern.

The group of Flat Crosscutting Patterns (Section 4.2) also presents moderate to large correlation with design stability. Data of Octopus in Tables 8-9 and 8-10, for instance, are used to illustrate this correlation. Octopus presented a correlation of 50% and 41% in MobileMedia Release 6 and Health Watcher Release 9, respectively. Like Communicative Concerns, the correlation of Flat Crosscutting Patterns and design stability in earlier releases is usually not better than weak. This fact suggests that crosscutting patterns might be ‘shaped’ during the system evolution. Therefore, they may not be reliable indicators of design stability in more unstable (earlier) releases.

Weak Correlation with Design Stability. The correlation between design stability and crosscutting patterns of the other two groups is not apparent. In fact, we could barely find a minor correlation for Inheritance-wise Concerns (Section 4.3) in some specific situations, such as MobileMedia Release 6. However, not even a minor correlation was found in Health Watcher. For example, data in Table 8-8 show 28% correlation of Hereditary Disease and module changes. On the other hand, we could not find any direct correlation in most other releases of MobileMedia or Health Watcher. Like Inheritance-wise Concerns, we could not find correlation of design stability and crosscutting patterns of the last group (Section 4.5). An exception is Behavioural Concern. Unlike the other patterns of this group, Tables 8-8 to 8-10 show that the correlation of Behavioural Concern and module changes is large. We believe that this fact happens because Behavioural Concerns present similar properties, such as scattering, of patterns in the group of Flat Crosscutting Patterns.

8.5 Threats to Validity

This section discusses some constraints to the study of this chapter. Similarly to the study of the previous chapter, the conclusions obtained here are restricted to the target software systems and analysed concerns. In other words, the impact of crosscutting patterns on design stability should not be directly generalised to other contexts. It is also important to mention that we have not investigated certain forms of design instability, such as the violation of design principles and anti-patterns. These instability symptoms are analysed by other authors [84, 150]. Instead, we focus only on actual component changes observed in a set of change scenarios.

Lozano, Wermelinger, and Nuseibeh [105] suggest that design problems appear and evolve over time. They argue that a single bad smell cannot be considered a threat by itself, but it may degenerate into a more harmful bad smell or it may promote the appearance of more bad smells. We believe this fact also occurs with the crosscutting patterns and it may explain why crosscutting patterns better correlate with design stability in later releases. However, we have not yet deeply analysed historical information of crosscutting patterns.

The evaluated systems might not be representative of industrial practice. To reduce this risk, we evaluated systems that come from heterogeneous application domains. The evaluated systems include a software product line, a Web-based information system, and a library of design patterns. Even though these are small to medium size applications, Health Watcher and MobileMedia are heavily based on industry-strength technologies, such as concurrency control, RMI, Servlets, JDBC, and API for mobile devices. In addition, these three systems were extensively used and evaluated in previous research work [56, 67, 73, 80].

One major threat to the study validity is the projection of concerns to design artefacts. In fact, we have observed in our studies that the concern projection into detailed design and implementation artefacts is even harder and more time-consuming than the projection of concern into architecture models. This observation is not a surprise due to the fact that more details about the solution are available at finer-grained decomposition levels. Fortunately, we run our controlled experiment with students in implementation artefacts (Section 6.2). Moreover, we observed that, for most of the concerns, the subjects projected concerns into source code with an accuracy of 80% or higher. Despite of these limitations, the exploratory study of this chapter provided useful insights about whether and which crosscutting patterns are good design stability indicators.

8.6 Summary

This chapter investigated whether crosscutting patterns and their detection means can be used as effective indicators of design stability. It reported the results of an exploratory study which detected crosscutting patterns in object-oriented and aspect-oriented designs of the three target applications (Section 8.1): MobileMedia, Health Watcher, and the Design Pattern library. A total of 69 concerns of these applications – 14 of MobileMedia, 11 of Health Watcher, and 44 of design patterns – were projected into their respective designs. Our tool, ConcernMorph (Section 5.6), was used to detect the crosscutting patterns in these three applications. ConcernMorph operationalises the detection strategies and algorithms presented in Chapter 5.

Our results of this evaluation allowed us to document the most common crosscutting patterns in aspect-oriented and object-oriented systems (Section 8.3). Additionally, they indicated crosscutting patterns that are not easily addressed by typical aspect-oriented programming (AOP) mechanisms, as available in AspectJ [2, 94]. This result may be useful to guide programming languages developers in the design of more effective AOP languages.

Our evaluation also correlated specific crosscutting patterns with stability of design components (Section 8.4.3). We found out that some crosscutting patterns, such as the ones in the groups of Communicative Concerns and Flat Crosscutting Patterns, are strong indicators of change-prone components in design models. However, this correlation is not so evident in crosscutting patterns of the other two groups. Since our results are restricted to the selected concerns (Section 8.1), further analyses may be required to confirm or refute our findings. The next chapter summarises this document and points out directions for future work.

9. Conclusions and Future Work

Building stable software is a challenging task mainly because designers need to simultaneously reason and make decisions about the modularisation of multiple concerns. These decisions drive the software development, from the architectural and detailed designs to implementation and maintenance stages. Hence, systematic assessment is essential to support designers building stable designs. Metrics and heuristic analysis are traditionally the fundamental mechanisms for assisting the assessment of design modularity and stability [29, 95, 111]. A plethora of software metrics and metric-based heuristic rules have been proposed for assessing design stability. Most of these metrics are defined upon the notion of module and are targeted at quantifying traditional modularity attributes, such as coupling and cohesion. However, it is hard to quantify the impact of non-modularised concerns without proper concern-sensitive indicators, such as concern metrics [46, 102, 151].

Research on concern analysis is still limited [129]. There are some concern metrics available in the literature [41, 46, 102, 151], but they are not properly formalised which makes them difficult to be composed in design heuristic rules. In particular, existing concern metrics quantify a few concern properties, such as scattering and tangling, but other concern dimensions are still uncovered. This is a deep problem to software engineers as recent studies have found that concern tangling and scattering are not decisive indicators to analyse design modularity and stability. Concerns manifest themselves in many different ways and they can cause varying levels of harm to design stability.

In this context, specific assessment techniques are required to identify the most harmful concern realisations. This thesis formalised 13 recurring patterns of crosscutting concerns, i.e., crosscutting patterns, identified in seven heterogeneous software systems. It also proposed a tool-supported heuristic technique, composed of concern metrics and heuristic analysis, to detect these crosscutting patterns. In an empirical evaluation, the crosscutting patterns were investigated and correlated with design stability. The next section details the key contributions of this PhD thesis and Section 9.3 points out directions for future work.

9.1 Contributions of the Thesis Re-Visited

We claimed in this document that there is a need to high level assessment mechanisms focusing on properties of crosscutting concerns to evaluate design stability. In this direction, we defined and formalised a set of 13 crosscutting patterns which have different impact on the stability of system designs. In other to formalise the crosscutting patterns, we proposed a concern-oriented conceptual framework which defined standard terminology and formalism for concern analysis. This conceptual framework was also used in this document to formalise a set of concern metrics and metrics-based heuristic strategies. The main goal of the formalised concern metrics and heuristic strategies was the detection of the crosscutting patterns. In summary, the five key contributions of this research can be stated as follows.

1. *Concern-Oriented Conceptual Framework*. The proposed conceptual framework defined a terminology and formalism for concern analysis. The terminology and formalism is abstract and language independent since the framework is intended to be generic and extensible. The framework is also composed of a comprehensive set of criteria for the comparison, evaluation, and definition of concern metrics. We demonstrated the framework usability and extensibility by formalising a number of concern metrics from different research groups. The framework was also applied in the formal definitions of the crosscutting patterns and their heuristic detection means.

2. *Set of Formalised Crosscutting Patterns*. The key contribution of this thesis was a list of crosscutting patterns formalised according to the proposed conceptual framework. These crosscutting patterns were described and classified based on an intuitive vocabulary that facilitates their recognition by software engineers. Moreover, they complement and refine previous considerations regarding the problems of dealing with crosscutting concerns [13, 41, 44, 74, 79, 109]. In particular, the proposed crosscutting patterns were inspired by the work of Ducasse, Girba, and Kuhn [41]. These authors defined the Black Sheep and Octopus crosscutting patterns based on the Distribution Map visual notation. Our contribution in this direction was the formalisation of these two crosscutting patterns and the definition of eleven novel ones.

3. *Heuristic Strategy for Detecting Crosscutting Patterns*. The third contribution of this document was a suite of concern-sensitive heuristic strategies, which aimed at detecting the proposed crosscutting patterns. The heuristic strategies also offer enhancements over traditional metrics-based assessment approaches. This suite can be extended and, by no means, we claimed that the proposed strategies are exhaustive. For instance, we discussed in Section 9.3.2 how the new heuristic strategies could be elaborated to detect additional concern-related design flaws. Our evaluation indicated promising results in favour of the heuristic strategies for determining concerns that should be aspectised, i.e., the ones that manifest harmful crosscutting patterns. It also pointed out that the heuristic analysis has a precision of about 80% for this aim (Section 6.3.3).

4. *Supporting Tool*. To effectively employ our concern heuristic assessment technique, it is imperative to provide automated support. We designed and implemented a concern assessment tool, called ConcernMorph. ConcernMorph aims at applying concern metrics and heuristic strategies to detect the crosscutting patterns. The tool was implemented as an Eclipse plugin [47] and reused an existing open source application, called ConcernMapper [33]. ConcernMapper [33] fulfilled some requirements for the conception of our tool. For instance, it provided means of projecting concerns into implementation artefacts. The goal of reusing an existing application was to speed up the implementation and to improve the tool reliability.

5. *Empirical Evaluation Focusing on Design Stability*. We designed and executed a complementary set of empirical and controlled experiments not only to corroborate the concern measurement framework but also to evaluate the elements of our heuristic assessment technique.

    a. *Controlled Experiment on Concern Identification*. We were aware of limitations of existing third-party concern identification tools which our work relies upon. So, to quantify the ability of developers to accurately characterise concerns in design artefacts, we performed a specific controlled experiment. The experiment hypothesis complemented our research questions and was stated as follows: "*the identification of system concerns in implementation artefacts does not* *depend on individual differences between developers*". Additionally, the research goal was to assess the accuracy of concern identification by students with varying levels of experience in the use of aspect-oriented techniques. We run this experiment in 3 classes composed of 8, 13, and 9 undergraduate and post-graduate students.

b. *Correlation of Crosscutting Patterns and Design Stability*. In the final evaluation of our concern assessment technique, we applied it to two medium size applications (MobileMedia [56] and Health Watcher [73]). This evaluation step aimed at assessing whether design instability can be predicted based on the manifested crosscutting patterns. Many successive releases of the target systems were investigated as well as their evolving concerns.

9.2 Lessons Learned

Chapter 1 of this thesis defined the problem, hypothesis, and research questions which underlined our research work. In particular, we aimed to verify the hypothesis that *some specific crosscutting patterns are more harmful to design stability than others*. We then observed in Chapter 2 that the current state of the art lacks means to verify or refute this hypothesis. Therefore, we presented our solution which is composed of formalism for concern analysis (Chapter 3), a set of formally defined crosscutting patterns (Chapter 4), and a tool-supported technique to detect these crosscutting patterns (Chapter 5). Finally, we designed and performed a set of empirical evaluation in Chapters 6 to 8 in order to verify the previously stated hypothesis.

This section reports some lessons learned based on our experience in detecting and analysing crosscutting patterns in design and architecture models. In our first evaluation step, we performed a controlled experiment to quantify the ability of developers to accurately identify and characterise concerns in design artefacts. Our experiment provided evidences that concerns can usually be projected into design artefacts with high degree of precision. Based on this result, we recommend the use of concern metrics and concern analysis in the identification of sources of design instability. We then investigated the two research questions (RQ.1 and RQ.2) derived from our hypothesis and presented in Chapter 1.

RQ.1. Does a high number of crosscutting patterns impact positively, negatively or have no impact on design stability?

To answer this question, we relied on two evolving software systems to assess whether design instability can be predicted based on the number of manifested crosscutting patterns. Both architecture and design models were used in the second evaluation step. First, we observed that crosscutting patterns are usually stronger indicators of design instabilities at the design level than they do at the architectural level. Moreover, we found that the more crosscutting patterns a single module has, the more unstable this module is likely to be. Therefore, our empirical analysis suggests that the answer for this question is "yes, a high number of crosscutting patterns impacts negatively on the stability of a software design".

RQ.2. If crosscutting patterns impact on design stability, which ones are better indicators of software stability?

Strong Correlation with Design Stability. Relying on the same two software systems, we also investigate the second research question (RQ.2 above) formulated in Chapter 1. In this case, we identified which crosscutting patterns are good indicators of change-prone classes and which ones are not. More precisely, we observed that the group of Communicative Concerns, except Tsunami, includes the crosscutting patterns which better indicate sources of design instability both at architecture and at design levels. Tsunami presented controversial results. It has shown large correlation with design stability in later releases of detailed design, but not in architecture models (Sections 7.5 and 8.4). The group of Flat Crosscutting Patterns follows closely the Communicative Concerns and also presents good correlation with module changes.

Weak Correlation with Design Stability. On the other extreme, we found in our empirical evaluation that in general the group of Inheritance-wise Concerns and the other crosscutting patterns do not present high correlation with design stability. However, some exceptions could be observed in specific situations. For instance, the Behavioural Concern crosscutting pattern presented moderate to large correlation with module changes in detailed design models both in earlier and later releases. The crosscutting patterns related to code clone, Copy Cat and Dolly Sheep, also presented moderate to large correlation with design stability in some releases. For instance, Copy Cat has shown 69% and 43% correlation degrees with module changes in the architecture model and in detailed design of MobileMedia Release 04, respectively (see Appendix E). Despite these special cases, all crosscutting patterns in these two groups have shown none or weak correlation with design stability.

9.3 Future Work

Future work could be driven by at least five main topics briefly discussed in this section. First, Section 9.3.1 introduces the application of aspect-oriented refactoring techniques to modularise concerns matching the crosscutting patterns. Second, Section 9.3.2 examines how concern analysis, such as concern-sensitive detection strategies, can be used to detect other design flaws in the light of three bad smells [63, 118, 127]. This section also provides a comparison of concern-sensitive detection strategies (based on concern metrics) and traditional detection strategies [95, 111, 112]. Third, Section 9.3.3 discussed possible extensions of our tool, ConcernMorph, in order to improve the analysis of crosscutting patterns. Forth, the identification and formalisation of additional crosscutting patterns are emphasised in Section 9.3.4. Finally, Section 9.3.5 discusses further empirical investigations based on crosscutting patterns.

9.3.1 Refactoring of Crosscutting Patterns

Refactoring [63, 93, 122] are behaviour-preserving transformations over code units aiming at improving quality attributes of software design. They are an essential technique used to mitigate design flaws emerging during software development and maintenance [128]. Aspect-oriented programming [88] provides explicit constructs for improving the modularisation of otherwise crosscutting concerns. For instance, the crosscutting patterns (Chapter 4) are based on typical properties of crosscutting concerns, such as scattering and tangling. Additionally, several works [10, 69, 78, 79, 83, 118] have been proposed to apply aspect-oriented refactoring with the goal of modularising crosscutting concerns. This kind of refactoring takes into account typical constructs of aspect-oriented programming, such as pointcut, advice, and intertype declaration [87, 88]. Therefore, it seems natural to consider crosscutting patterns as symptoms to motivate the application of aspect-oriented refactoring.

We are currently working on the definitions of aspect-oriented refactoring for the crosscutting patterns[9]. To illustrate this refactoring technique, this section briefly presents aspect-oriented refactoring aiming at modularising the three flat crosscutting patterns (Section 4.2), namely Black Sheep, Octopus, and God Concern. We focus these crosscutting patterns because they better capture the key properties of crosscutting concerns, *scattering* and *tangling*, addressed by aspect-oriented languages. That is, most aspect-oriented mechanisms aim at reducing these key concern properties [88, 114]. Details of the refactoring steps are described in Appendix F.

Black Sheep Refactoring. The goal of this refactoring is to better encapsulate concerns classified as Black Sheep. The abstract representation of this refactoring is presented in Figure 9-1 (top). First, it identifies classes implementing parts of the Black Sheep concern and, then, it modularises those parts into aspects. Using our terminology of crosscutting patterns, each class is called *sheep* (or black sheep) and the whole system is a *herd*. Alternatively, each part of a class realising the target concern is called a *sheep slice*.

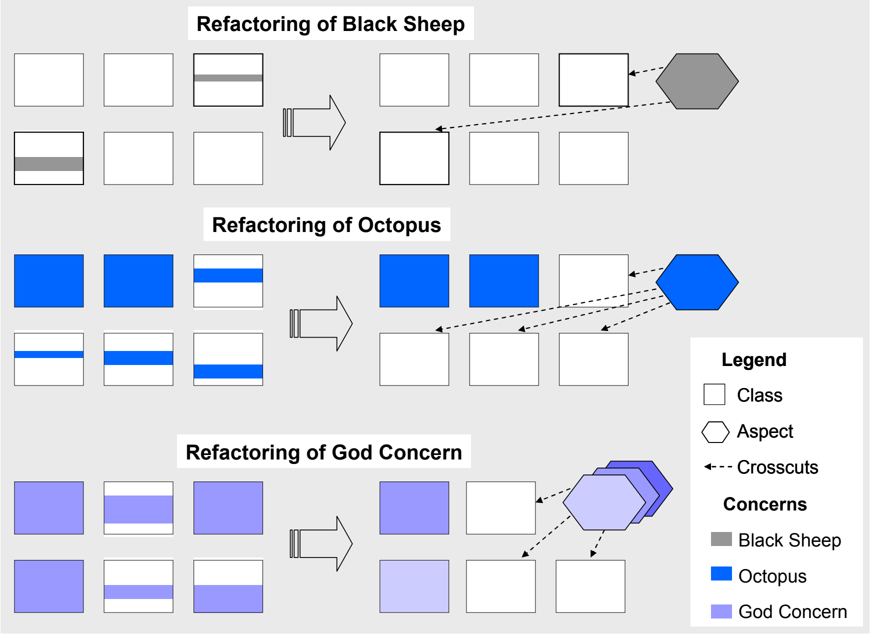


Figure 9-1: Aspect-oriented refactoring for flat crosscutting patterns.

[9] Refactoring strategies presented in this section have been defined in cooperation with Bruno Silva, a Master student at Federal University of Rio Grande do Sul.

Octopus Refactoring. The goal of this refactoring is to better modularise concerns classified as Octopus. The body and tentacles are the main characteristics of this crosscutting pattern. Hence, the refactoring steps aim at moving to aspects parts of code composing the body or touched by tentacles of an Octopus concern. Figure 9-1 (middle) shows the abstract representation of this refactoring. This figure shows that the Octopus' tentacles have to be separated using aspects while refactoring the Octopus' body is not mandatory.

God Concern Refactoring. The goal of this refactoring is to modularise self-contained portions of God Concern. The existence of a concern with characteristics of God Concern reveals several design modularity problems. In addition to being scattered and tangled over many components, the concern also concentrates multiple intentions and functionalities. Figure 9-1 (bottom) presents an abstract representation of this refactoring. The key action to address problems of this crosscutting pattern is to try to decompose the concern and, if possible, to distribute unrelated intentions over new sub-concerns. Then, each sub-concern is modularised by means of aspect-oriented techniques.

9.3.2 Detection of Additional Concern-Aware Bad Smells

This section describes our ongoing work on the application of concern-sensitive detection strategies (Chapter 5) to detect well-known design flaws, such as bad smells [63, 127]. We used three bad smells, Feature Envy [63], Shotgun Surgery [63], and God Class [127], to illustrate the use of our heuristic technique. Feature Envy is related to the misplacement of a piece of code, such as an operation or an attribute, realising a concern. That is, the concern seems more interested in a component other than the one it actually is in [63]. Shotgun Surgery, on the other hand, occurs when a change in a characteristic (or concern) of the system implies many changes to a lot of different places [63]. God Class refers to a component (class or aspect) which performs most of the work, delegating only minor details to a set of trivial classes and using data from them [127]. We selected these three bad smells because (i) previous work related them to crosscutting concerns [127] and (ii) they are representatives of three different groups of design flaws discussed by Marinescu [111]. Hence, by addressing different categories of design flaws, we can better correlate our results with Marinescu's studies [111].

Condition C - *High Inter-Component Coupling*:
(CSC / CBC) > ((NOCA+CDO) / (NOA+NOO))

Condition D - *Low Intra-Component Coupling*:
(ICSC / ((NOA+NOO)-(NOCA+CDO))) < ((NOCA+CDO) / (NOA+NOO))

R14 - *Feature Envy*:
if (*High Inter-Component Coupling*) **and** (*Low Intra-Component Coupling*) **and** (LCC > 1)
then Crosscutting Concern is Feature Envy

R15 - *Shotgun Surgery*:
if Concern is (*Tangled*) **and** (*Highly Scattered*) **and** (*Crosscutting*)
then Crosscutting Concern is Shotgun Surgery

R16 - *God Class*:
if (LCC >2) **and** (NOA > (NOAsystem/NC)) **and** (NOO > (NOOsystem/NC))
then Component is a God Class

Figure 9-2: Detection strategies for bad smells

Figure 9-2 shows concern-sensitive detection strategies for detecting the three aforementioned bad smells. The first rule, R14, aims at detecting Feature Envy and uses a combination of concern metrics, coupling metrics, and size metrics. The used concern metrics were defined in Section 3.3. All metrics were also briefly described in Tables 5-1 and 5-2 (Section 5.1). To be considered Feature Envy, a crosscutting concern has to satisfy the two conditions appearing in R14, i.e., Condition C (*High Inter-Component Coupling*) and Condition D (*Low Intra-Component Coupling*). The concern has high inter-component coupling when its percentage of coupling (CSC/CBC) is higher than the percentage of internal component members that realise this concern ((NOCA+CDO) / (NOA+NOO)). In other words, the ratio of coupling to size (measured in terms of attributes and operations) of such crosscutting concern is higher than the same ratio for all other concerns in the same component. Similar computation is performed for identifying low intra-component coupling.

The remaining two detection strategies for bad smells (Figure 9-2), R15 and R16, are intended to detect Shotgun Surgery and God Class, respectively. Differently from all other detection strategies, R15 is composed of outcomes from other rules. More precisely, a concern is classified as Shotgun Surgery if it was previously identified as Tangled (R02), Highly Scattered (R04), and Crosscutting (R08); these rules were presented in Section 5.1. Finally, a component is flagged as God Class (R16) if, in addition to high number of internal members (attributes and operations), it also realises many concerns. R16 distinguishes high and low numbers of internal members by comparing them with the average size of all other system components (NOAsystem/NC and NOOsystem/NC). NOAsystem and NOOsystem indicate the number of operations and attributes of a system, respectively. Furthermore, this last rule uses a threshold value of 2 for the metric Lack of Concern-based Cohesion (LCC). The reasoning is that a component implementing more than two concerns (LCC > 2) tends to aggregate too much unrelated responsibility. Two concerns in a component may not be a problem, e.g., the roles of a design pattern discussed in Section 6.3.3.

In order to evaluate these concern-sensitive detection strategies for bad smell detection, we applied the rules R14 to R16 (Figure 9-2) to identify the Shotgun Surgery [63], Feature Envy [63], and God Class [127] bad smells in six systems. The target systems were introduced and used the analysis of Section 6.3. We also independently applied the traditional detection strategies proposed by Marinescu [95, 111] for detecting the same bad smells. The application of each rule pointed out design fragments suspect of having one of the three bad smells. We then compared the results from the two suites of rules (concern-sensitive vs. traditional) by undertaking a manual inspection in all suspect design fragments. In circumstances where the existence of a bad smell was not clear about, we contacted specialists and researchers with long-term experience on the development and assessment of the target designs. The manual inspection allowed us to verify whether the suspect design fragments were indeed affected by the design flaw.

After this inspection, the total suspect design fragments for each bad smell were classified in two categories: (i) ‘Hits’: those heuristically suspect fragments that have confirmed to be affected by the bad smell, and (ii) ‘False Positives’: those heuristically suspect fragments that revealed not to be affected by the bad smell. Table 9-1 summarises the results of applying both concern-sensitive and traditional detection strategies. This table also shows the total number of hits and false positives (FP) for each bad smell and the percentage (under brackets) with respect to the total number of suspect fragments. Note that the results in Table 9-1 are given in different view points. In the concern-sensitive rules, the values for Shotgun Surgery and Feature Envy represent the number of concerns (since R14 and R15 classify concerns). On the other hand, God Class and traditional rules present the results in terms of the number of classes or operations.

Table 9-1: Statistics for bad smell detection

Bad Smell	**Concern-Sensitive Rules**		**Traditional Rules**	
	Hits (%)	**FP (%)**	**Hits (%)**	**FP (%)**
Shotgun Surgery	8 (89%)	1 (11%)	9 (56%)	7 (44%)
Feature Envy	3 (100%)	0 (0%)	1 (17%)	5 (83%)
God Class	8 (80%)	2 (20%)	1 (17%)	5 (83%)

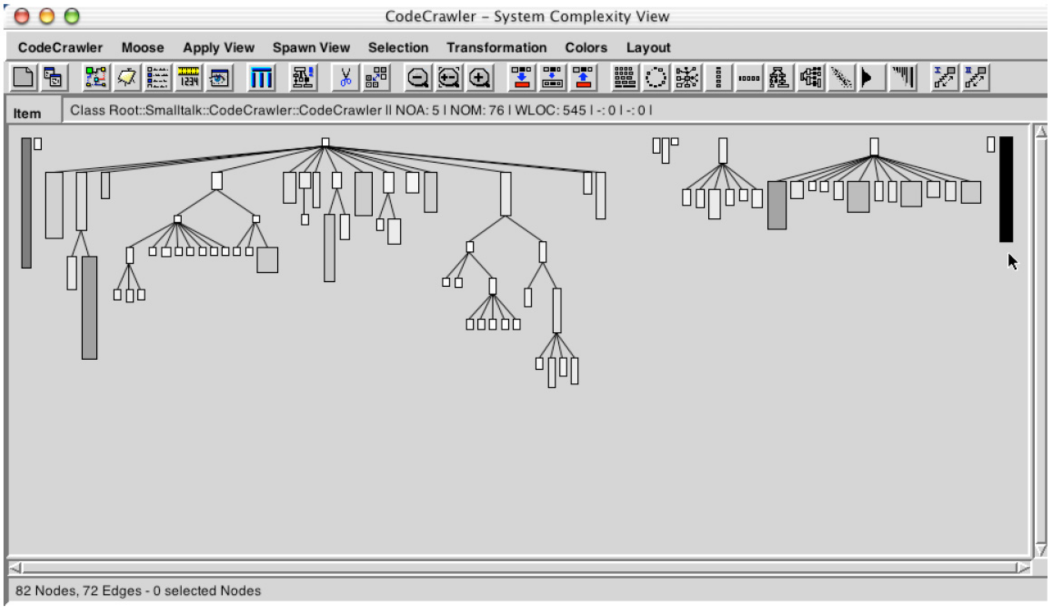
The results of Table 9-1 show that concern-sensitive detection strategies presented superior accuracy than traditional rules for detecting all three studied bad smells. The former presented no more than 20% of false positives, whereas the latter exhibited many false positives. This observation is not a surprise because we selected bad smells that are closely related to separation of concerns. For this reason, we could also verify that the advantage in favour of our rules was mainly due to the fact that they are sensitive to the design concerns. For instance, many false positives of the traditional rule for Shotgun Surgery occurred because its composing metrics do not distinguee coupling between classes of the same concern and coupling between classes of different concerns. Further investigations are required to confirm or to refute our preliminary findings.

9.3.3 Improved Tool Support

As part of this PhD research, we implemented ConcernMorph, a prototype tool for detecting the crosscutting patterns based on concern metrics and heuristic analysis. The usability of ConcernMorph was evaluated by analysing code bases of three software systems MobileMedia, Health Watcher, and a Design Pattern library. However, some improvements are desirable to foster the widely adoption of ConcernMorph in practice. For instance, the tool could be extended to incorporate more sophisticated means for mining and mapping concerns to design elements. Exception Handling is an example of concern which can easily be automatically detected in Java systems since it is clearly marked by the language constructs, such as try-catch blocks. On the other hand, concerns like Persistence, Distribution, and Concurrency are more application dependents leading to a more complex identification process. The identification of these concerns in ConcernMorph would benefit from elaborated mining support.

Moreover, software systems are nowadays developed using several languages. It is easy to find examples of large software systems which are designed and implemented using a combination of different languages, such as Java, C++, and PHP. However, the current implementation of ConcernMorph analyses just Java code. Therefore, additional modelling and programming languages could be incorporated in order to allow users analysing this category of multi-language systems.

ConcernMorph also supports limited visualisation interfaces. However, as the target application evolves and the number of concerns increases, analysis of crosscutting patterns becomes a challenge and time-consuming task. Therefore, another further extension of ConcernMorph could incorporate visual representations of crosscutting patterns using software visualisation techniques [24, 25, 41, 96, 97, 139]. For instance, Distribution Map proposed by Ducasse, Girba, and Kuhn [41] is a generic technique that can be used to visualise and analyse the association of concerns and components (see Figure 2-4 in Section 2.4). This visualisation technique is particularly useful to visualise crosscutting patterns in the category of Flat Crosscutting Patterns (Section 4.2). In addition, Polymetric Views [14] include a set of two-dimensional visual representations for analysing software properties. Polymetric Views use rectangles to represent software entities, such as classes and interfaces, and edges to represent inheritance relationships between those entities.



Class Root::Smalltalk::CodeCrawler::CodeCrawler || NOA: 5 | NOM: 76 | WLOC: 545 | -: 0 | -: 0 |

82 Nodes, 72 Edges - 0 selected Nodes

Figure 9-3: A Polymetric View representation of inheritance trees [96].

Figure 9-3 shows a Polymetric View model which represents inheritance trees. We believe that the use of Polymetric Views [96, 97] could be used to enhance the visual representation of crosscutting patterns in the category of Inheritance-wise Concerns (Section 4.3) and Communicative Concerns (Section 4.4).

A key advantage of these visual notations, such as Distribution Map and Polymetric Views, is that they are scalable. That is, they can represent hundreds or even thousands of software components in a single screen. Moreover, there are current available tools and APIs to support them. For instance, CodeCrawler [96] allows users to browse the views. Users can also use filters and search queries based, for instance, on the component names and measured attributes. In CodeCrawler, filtering and searching mechanisms affect all of views simultaneously. In this context, the integration of CodeCrawler and ConcernMorph would provide a powerful environment to analyse the system concerns and crosscutting patterns.

9.3.4 Definition and Formalisation of Additional Crosscutting Patterns

In Chapter 4, we identified and formalised thirteen crosscutting patterns capturing characteristics of crosscutting concerns that have been found to occur widely in a number of different systems. These patterns capture the fact that crosscutting concerns may manifest themselves in significantly different ways, ranging from a few minor elements of the concern scattered across a single object hierarchy, to many elements littered across the entire system. We have shown in Chapters 7 and 8 that there are correlations between certain types of crosscutting patterns and design stability. In particular, some crosscutting patterns are more harmful than others.

A natural question is how complete the proposed suite of crosscutting patterns is. By no means have we claimed that the proposed suite is exhaustive. In fact, the proposed crosscutting patterns act as a stepping stone towards the understanding of particular ‘shapes’ of crosscutting concerns and their correlations with design quality attributes. Therefore, new definitions of crosscutting patterns are mostly welcome. These new crosscutting patterns can rely (or not) on the concern terminology and formalism introduced in Chapter 3. The key point is that they should be defined in an abstract way in order to be instantiated for different modelling and programming languages. The proposed quantitative approach can also be extended in a number of ways to support the detection of new crosscutting patterns. For instance, new concern metrics can be formalised using the proposed conceptual framework (Chapter 3). Moreover, the proposed heuristic assessment technique and its supporting tool (Chapter 5) can be extended to support the detection of future crosscutting patterns.

New crosscutting patterns can be defined either individually or by composing existing ones. Dolly Sheep (Section 4.5) is an example of a crosscutting pattern composed from the definitions of two others (Black Sheep and Copy Cat). In addition to the analysis we performed in Chapters 7 and 8, further analysis of typical combinations of the crosscutting patterns and the positive and negative impacts of these combinations in the system design are required. For instance, in our empirical investigation, we found out that some crosscutting concerns classified as Climbing Plant also present the Octopus pattern. This case is illustrated in Figure 9-4 where the `Cloneable` interface is completely dedicated to the concern realisation while the classes are crosscut by the concern. Someone may want to consider this Climbing Plant and Octopus combination as a different crosscutting pattern. Let's call it *Seaweed* for illustration.

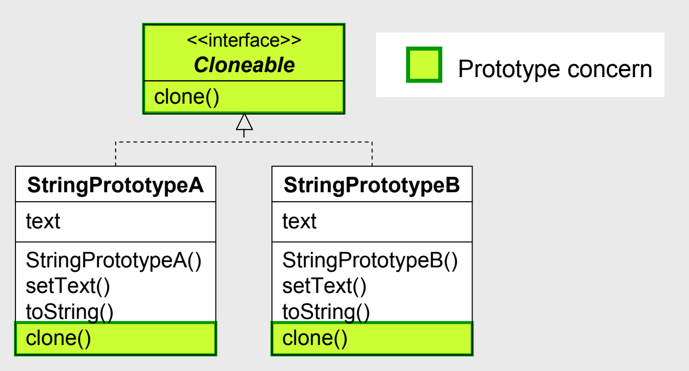


Figure 9-4: The Prototype design pattern as Seaweed.

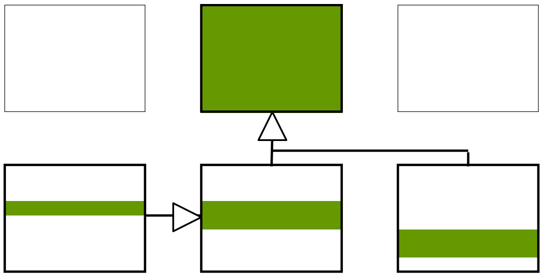


Figure 9-5: Abstract representation of Seaweed.

Figure 9-5 presents the abstract representation of Seaweed using the same notation adopted for the other crosscutting patterns. The formalisation of Seaweed can be straightforwardly defined since it is a composition of previously defined crosscutting patterns. The Seaweed formal definition presented below states that a concern is Seaweed if, and only if, this concern is both Climbing Plant and Octopus. That is, the two non-empty sets of components which define the Climbing Plant and Octopus crosscutting patterns are equal. The formal definitions of Octopus and Climbing Plant are presented earlier in Sections 4.2 and 4.3, respectively.

$$\text{Seaweed}(con) \Leftrightarrow \text{Octopus}(con) = \text{ClimbingPlant}(con) \neq \emptyset$$

9.3.5 Further Evaluation

The proposed heuristic assessment technique based on crosscutting patterns has been evaluated using representative systems from different domains. In particular, the quantitative studies presented in Chapters 7 and 8 provide considerable evidence of the correlation of some crosscutting patterns and design stability. However, it is necessary to undertake additional studies in order to assess the usefulness of analysing crosscutting patterns in different contexts. For instance, we foresee the following directions for further empirical investigations.

- Our controlled experiment with students (Section 6.2) evaluated the ability of developers to project concerns into the source code. We relied on 6 concerns of one application in this experiment. Moreover, four of these concerns were non-functional requirements. Interestingly, one concern which represented a functional requirement, Business, turned out to be the hardest one to be projected. Further experimentation relying on additional concerns and applications are required to verify, for instance, whether functional requirements are indeed more difficult to be projected.

- It is desirable to uncover sources of design stabilities earlier in the software development process. Unfortunately, our studies based on architecture models indicated weak correlation of most crosscutting patterns and architectural stability. Further investigations could be performed in this direction to, hopefully, contradict our initial findings.

- In our studies we focus on the assessment of designs with respect to their evolutionary module changes. However, other means of assessing design stability, such as code clones [104], bad smells [105], and design principle violations [150], can also be taken into consideration. For instance, Lozano and Wermelinger [104] suggest that having a clone may increase the maintenance effort of changing a method. In a different work, they also investigate the relationship between bad smells and design principle violations in order to better identify the root causes of design instabilities [105]. Moreover, Wermelinger, Yu, and Lozano [150] performed an empirical study relying on structural design principles to assess the maintenance effort of software architecture. These different facets of design stability can be obviously used in future work as alternative indicators to module changes. In particular, they are useful when the software engineer has only on release of the target system at hand.

- Since the crosscutting patterns are defined in a language-independent way, they can be used for comparing design alternatives. For instance, the crosscutting patterns could be used to compare designs of the same system built on the basis of different design and architectural patterns. This kind of comparative analysis is particularly important to support the selection of a component among several alternatives in a component-based software development [144].

- Eaddy *et al*. [44] have recently carried out an empirical study using three open-source Java projects [8] to assess the correlation of crosscutting concerns and error-proneness. Their study aimed at testing the hypothesis that the more scattered a concern's implementation is, the more likely it is to have defects. They found a moderate to strong correlation between concern scattering and error-proneness for the three target systems. However, further experiments are needed to answer an additional important question. If crosscutting concerns cause defects, which crosscutting patterns are better indicators of error-proneness?

- We investigated in Chapters 7 and 8 the correlation of specific crosscutting patterns and design stability. We verified that many crosscutting pattern instances in a component may lead to excessive changes in this component during the system evolution. However, we have not investigated the impact of specific combinations of crosscutting patterns on design stability. In fact, one concern classified according to a crosscutting pattern typically crosscuts the realisation of other concerns in different crosscutting patterns. In this situation, we can say that a crosscutting pattern instance is scattered and tangled to other crosscutting patterns through their participant classes. This crosscutting phenomenon of the patterns potentially has negative impacts on design quality attributes, such as design stability. Therefore, it is also important to empirically analyse combinations of crosscutting patterns and their impact on the system design.

In conclusion, this work is the first investigation to focus on the correlation of crosscutting patterns and design stability. And so, this thesis is a stepping stone towards more intentional assessment of the system designs based on their driven concerns. The thesis has achieved its objectives and initiated a new avenue in concern-based assessment of software designs. As an original work in this area, it has addressed some key issues, but also uncovered a number of further research topics.

References

[1] AJATO: an AspectJ Assessment Tool. Accessed: 30 Sept 2009, URL http://www.teccomm.les.inf.puc-rio.br/emagno/ajato/

[2] AspectJ Project. Accessed: 30 Sept 2009, URL http://www.eclipse.org/aspectj/

[3] P.F. Baldi, C.V. Lopes, E.J. Linstead, and S.K. Bajracharya, "A Theory of Aspects as Latent Topics," *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, Nashville, TN, USA: ACM, 2008, pp. 543-562.

[4] C.Y. Baldwin and K.B. Clark, *Design Rules, Vol. 1: The Power of Modularity*, The MIT Press, 2000.

[5] T.T. Bartolomei, A. Garcia, C. Sant'Anna, and E. Figueiredo, "Towards a Unified Coupling Framework for Measuring Aspect-Oriented Programs," *Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA)*, Portland, Oregon: ACM, 2006, pp. 46-53.

[6] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley Professional, 1997.

[7] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 1998, p. 368.

[8] C. Begin, B. Goodin, and L. Meadors, *iBATIS in Action*, Greenwich, CT, USA: Manning Publications Co., 2007.

[9] T.J. Biggerstaff, B.G. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," *Proceedings of the 15th International Conference on Software Engineering (ICSE)*, Baltimore, Maryland, United States: IEEE Computer Society Press, 1993, pp. 482-498.

[10] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, 2006, pp. 698-717.

[11] B. Boehm and V.R. Basili, “Software Defect Reduction Top 10 List,” *IEEE Computer*, vol. 34, 2001, pp. 135-137.

[12] G. Booch, *Object-Oriented Analysis and Design with Applications*, Addison-Wesley Professional, 1993.

[13] J. Boulanger and M.P. Robillard, “Managing Concern Interfaces,” *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, IEEE Computer Society, 2006, pp. 14-23.

[14] L.C. Briand, J.W. Daly, and J.K. Wüst, “A Unified Framework for Coupling Measurement in Object-Oriented Systems,” *IEEE Transactions on Software Engineering (TSE)*, vol. 25, 1999, pp. 91-121.

[15] L.C. Briand, J.W. Daly, and J. Wuest, “A Unified Framework for Cohesion Measurement,” *Proceedings of the 4th International Symposium on Software Metrics*, IEEE Computer Society, 1997, p. 43.

[16] L.C. Briand, S. Morasca, and V.R. Basili, “Measuring and Assessing Maintainability at the End of High Level Design,” *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 1993, pp. 88-97.

[17] R. Brooks, “Towards a Theory of the Comprehension of Computer Programs,” *International Journal of Man-Machine Studies*, vol. 18, Nov. 1983, pp. 543-554.

[18] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen, “An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns,” *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2004, pp. 209, 200.

[19] S. Bryton and F. Brito e Abreu, “Towards Paradigm-Independent Software Assessment,” *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2007, pp. 40-54.

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: a System of Patterns*, John Wiley & Sons, 1996.

[21] N. Cacho, F.C. Filho, A. Garcia, and E. Figueiredo, “EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming,” *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD)*, Brussels, Belgium: ACM, 2008, pp. 72-83.

[22] N. Cacho, C. Sant’Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena, “Composing Design Patterns: a Scalability Study of Aspect-Oriented Programming,” *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD)*, Bonn, Germany: ACM, 2006, pp. 109-121.

[23] N. Cacho, T. Batista, A. Garcia, C. Sant'Anna, and G. Blair, “Improving Modularity of Reflective Middleware with Aspect-Oriented Programming,” *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM)*, Portland, Oregon: ACM, 2006, pp. 31-38.

[24] G.D.F. Carneiro, R. Magnavita, and M. Mendonça, “Combining Software Visualization Paradigms to Support Software Comprehension Activities,” *Proceedings of the 4th ACM Symposium on Software Visualization (SoftViz), demo session*, Ammersee, Germany: ACM, 2008, pp. 201-202.

[25] G.D.F. Carneiro, R.C. Magnavita, E. Spinola, F. Spinola, and M. Mendonca, “Evaluating the Usefulness of Software Visualization in Supporting Software Comprehension Activities,” *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Kaiserslautern, Germany: ACM, 2008, pp. 276-278.

[26] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe, “A Qualitative Comparison of Three Aspect Mining Techniques,” *Proceedings of the 13th International Workshop on Program Comprehension (IWPC)*, IEEE Computer Society, 2005, pp. 13-22.

[27] M. Ceccato and P. Tonella, “Measuring the Effects of Software Aspectization,” *Proceedings of the 1st Workshop on Aspect Reverse Engineering*, 2004.

[28] C. Chavez, C. Lucena, “A Metamodel for Aspect-Oriented Modeling”. *Proceedings of the Workshop on Aspect-oriented Modeling with UML, 1st International Conference on Aspect-Oriented Software Development*, 2002.

[29] S.R. Chidamber and C.F. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering (TSE)*, vol. 20, 1994, pp. 476-493.

[30] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional Requirements in Software Engineering*, Springer, 1999.

[31] P. Clements, R. Kazman, and M. Klein, Evaluating Software Architectures: Methods and Case Studies, Addison Wesley, 2001.

[32] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional, 2001.

[33] ConcernMapper: Simple Separation of Concerns for Eclipse. Accessed: 30 Sept 2009, URL http://www.cs.mcgill.ca/~martin/cm/

[34] J. Conejero, E. Figueiredo, A. Garcia, J. Hernandez, and E. Jurado, “Early Crosscutting Metrics as Predictors of Software Instability,” *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS)*, 2009.

[35] Crosscutting Patterns and Design Stability (*supplementary website*): 30 Sept 2009, URL http://www.lancs.ac.uk/postgrad/figueire/concern/patterrns/

[36] CVS Plug In. Accessed: 30 Sept 2009, URL http://www.eclipse.org/eclipse/platform-cvs/

[37] B. Dagenais, S. Breu, F.W. Warr, and M.P. Robillard, “Inferring Structural Patterns for Concern Traceability in Evolving Software,” *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Atlanta, Georgia, USA: ACM, 2007, pp. 254-263.

[38] H. Deitel and P. Deitel, *Java How to Program*, Prentice Hall, 2004.

[39] Detecting Architecture Instabilities with Concern Traces: 30 Sept 2009, URL http://www.comp.lancs.ac.uk/~shakilkh/MMData/

[40] E. Dijkstra, *A Discipline of Programming*, Prentice Hall, Inc., 1976.

[41] S. Ducasse, T. Girba, and A. Kuhn, “Distribution Map,” *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2006, pp. 203-212.

[42] S. Ducasse, M. Rieger, and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 1999, p. 109.

[43] A.H. Dutoit, R. McCall, I. Mistrik, and B. Paech, *Rationale Management in Software Engineering*, Springer, 2006.

[44] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan, and A. Aho, “Do Crosscutting Concerns Cause Defects?,” *IEEE Transactions on Software Engineering (TSE)*, vol. 34, 2008, pp. 515, 497.

[45] M. Eaddy, A. Aho, G. Antoniol, and Y. Guéhéneuc, “CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis,” *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, IEEE Computer Society, 2008, pp. 53-62.

[46] M. Eaddy, A. Aho, and G.C. Murphy, “Identifying, Assigning, and Quantifying Crosscutting Concerns,” *Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques (ACoM)*, IEEE Computer Society, 2007, p. 2.

[47] Eclipse Project. Accessed: 30 Sept 2009, URL http://www.eclipse.org/

[48] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, “Does Code Decay? Assessing the Evidence from Change Management Data,” *IEEE Transactions on Software Engineering (TSE)*, vol. 27, 2001, pp. 1-12.

[49] M.O. Elish and D. Rine, “Investigation of Metrics for Object-Oriented Design Logical Stability,” *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society, 2003, p. 193.

[50] M. Fayad, “Accomplishing Software Stability,” *Communications of the ACM*, vol. 45, 2002, pp. 111-115.

[51] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, Pws Pub Co, 1996.

[52] E. Figueiredo, B. Silva, C. Sant'Anna, A. Garcia, J. Whittle, and D. Nunes, "Crosscutting Patterns and Design Stability: An Exploratory Analysis," *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*, 2009.

[53] E. Figueiredo, J. Whittle, and A. Garcia, "ConcernMorph: Metrics-based Detection of Crosscutting Patterns," *Proceedings of the 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, demo session, 2009.

[54] E. Figueiredo, I. Galvao, S. Khan, A. Garcia, C. Sant'Anna, A. Pimentel, A. Medeiros, L. Fernandes, T. Batista, R. Ribeiro, P. van den Broek, M. Aksit, S. Zschaler, and A. Moreira, "Detecting Architecture Instabilities with Concern Traces: An Exploratory Study," *Proceedings of the Joint 8th Working IEEE/IFIP Conference on Software Architecture (WICSA) and the 3rd European Conference on Software Architecture (ECSA)*, 2009.

[55] E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena. "Applying and Evaluating Concern-Sensitive Design Heuristics". *Proceedings of the 23rd Brazilian Symposium on Software Engineering (SBES)*. Fortaleza, 2009.

[56] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F.C. Filho, and F. Dantas, "Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability," *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany: ACM, 2008, pp. 261-270.

[57] E. Figueiredo, C. Sant'Anna, A. Garcia, T. Bartolomei, W. Cazzola, and A. Marchetto, "On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework," *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, 2008, pp. 183-192.

[58] E. Figueiredo, A. Garcia, and C. Lucena, “AJATO: an AspectJ Assessment Tool,” *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, demo section, 2006.

[59] E. Figueiredo, A. Garcia, C. Sant’Anna, U. Kulesza, and C. Lucena, “Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method,” *Workshop on Quantitative Approaches in OO Software Engineering (QAOOSE)*, 2005.

[60] F.C. Filho, N. Cacho, E. Figueiredo, A. Garcia, C. Rubira, J. Amorim, and H. Silva, “On the Modularization and Reuse of Exception Handling with Aspects,” *Software: Practice and Experience (SP&E)*, 2009.

[61] F.C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. Rubira, “Exceptions and Aspects: the Devil is in the Details,” *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Portland, Oregon, USA: ACM, 2006, pp. 152-162.

[62] R.E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-Oriented Software Development*, Addison-Wesley Professional, 2004.

[63] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.

[64] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[65] A.F. Garcia, E. Figueiredo, C.N. Sant'Anna, M. Pinto, and L. Fuentes, “Representing Architectural Aspects with a Symmetric Approach,” *Proceedings of the 15th Workshop on Early Aspects at AOSD*, Charlottesville, Virginia, USA: ACM, 2009, pp. 25-30.

[66] A. Garcia, C. Sant’Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, “Modularizing Design Patterns with Aspects: A Quantitative Study,” *Transactions on Aspect-Oriented Software Development I*, 2006, pp. 36-74.

[67] A. Garcia, C. Sant’Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, “Modularizing Design Patterns with Aspects: a Quantitative Study,”

Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD), Chicago, Illinois: ACM, 2005, pp. 3-14.

[68] A. Garcia, C. Sant'Anna, C. Chavez, V.T. da Silva, C.J. de Lucena, and A. von Staa, "Separation of Concerns in Multi-agent Systems: An Empirical Study," *Proceedings of the workshop on Software Engineering for Multi-Agent Systems (SELMAS)*, 2004, pp. 343-344.

[69] V. Garcia, E. Piveta, D. Lucredio, A. Alvaro, E. Almeida, A. Prado, and L. Zancanella, "Manipulating Crosscutting Concerns," *Proceedings of the 4th Latin American Conference on Patterns Languages of Programming (SugarLoafPlop)*, 2004.

[70] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 2002.

[71] O. Gotel and C. Finkelstein, "An Analysis of the Requirements Traceability Problem," *Proceedings of the 1st International Conference on Requirements Engineering (RE)*, 1994, pp. 94-101.

[72] P. Greenwood, A. Garcia, A. Rashid, E. Figueiredo, C. Sant'Anna, N. Cacho, A. Sampaio, S. Soares, P. Borba, M. Dosea, R. Ramos, U. Kulesza, T. Bartolomei, M. Pinto, L. Fuentes, N. Gamez, A. Moreira, J. Araujo, T. Batista, A. Medeiros, F. Dantas, L. Fernandes, J. Wloka, C. Chavez, R. France, and I. Brito, "On the Contributions of an End-to-End AOSD Testbed," *Proceedings of the Early Aspects at ICSE: Workshops in Aspect-Oriented Requirements Engineering and Architecture Design*, IEEE Computer Society, 2007, p. 8.

[73] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid, "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2007, pp. 176-200.

[74] W.G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Software*, vol. 23, 2006, pp. 51-60.

[75] W.G. Griswold, J.J. Yuan, and Y. Kato, “Exploiting the Map Metaphor in a Tool for Software Evolution,” *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 265-274.

[76] J.V. Gurp and J. Bosch, “Design Erosion: Problems and Causes,” *Journal of Systems and Software (JSS)*, vol. 61, 2002, pp. 105-119.

[77] Y. Han, G. Kniesel, A.B. Cremers, “Towards Visual AspectJ by a Meta Model and Modeling Notation”. *Proceedings of the Workshop on Aspect-Oriented Modeling*, 2005.

[78] S. Hanenberg, C. Oberschulte, and R. Unland, “Refactoring of Aspect-Oriented Software”. *NetObject Days*. Erfurt, Germany, 2003.

[79] J. Hannemann, G.C. Murphy, and G. Kiczales, “Role-based Refactoring of Crosscutting Concerns,” *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, Illinois: ACM, 2005, pp. 135-146.

[80] J. Hannemann and G. Kiczales, “Design Pattern Implementation in Java and aspectJ,” *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, Washington, USA: ACM, 2002, pp. 161-173.

[81] W. Harrison, H. Ossher, S. Sutton, and P. Tarr, “Concern Modeling in the Concern Manipulation Environment,” *Proceedings of the 2005 workshop on Modeling and Analysis of Concerns in Software (MACS)*, St. Louis, Missouri: ACM, 2005, pp. 1-5.

[82] J. Hunter, *Java Servlet Programming*, O’Reilly, 1998.

[83] M. Iwamoto and J. Zhao, “Refactoring Aspect-Oriented Programs,” *Proceedings of the 4th AOSD Modeling With UML Workshop at UML’03*, San Francisco, CA, 2003.

[84] D. Kelly, “A Study of Design Characteristics in Evolving Software Using Stability as a Criterion,” *IEEE Transactions on Software Engineering (TSE)*, vol. 32, 2006, pp. 315-329.

[85] J. Kerievsky, *Refactoring to Patterns*, Addison Wesley, 2004.

[86] S.S. Khan, P. Greenwood, A. Garcia, and A. Rashid, “On the Impact of Evolving Requirements-Architecture Dependencies: An Exploratory Study,” *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE)*, Springer-Verlag, 2008, pp. 243-257.

[87] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “An Overview of AspectJ,” *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag, 2001, pp. 353, 327.

[88] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland: 1997.

[89] B. Kitchenham, S.L. Pfleeger, and N. Fenton, “Towards a Framework for Software Measurement Validation,” *IEEE Transactions on Software Engineering (TSE)*, vol. 21, 1995, pp. 929-944.

[90] R. Komondoor and S. Horwitz, “Using Slicing to Identify Duplication in Source Code,” *Proceedings of the 8th International Symposium on Static Analysis*, Springer-Verlag, 2001, pp. 40-56.

[91] D.C. Kozen, *The Design and Analysis of Algorithms*, 1st edition, Springer, 1991.

[92] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A.V. Staa, and C. Lucena, “Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study,” *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2006, pp. 223-233.

[93] R. Laddad, *Aspect Oriented Refactoring*, Addison-Wesley Professional, 2008.

[94] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, 2003.

[95] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, Springer, 2006.

[96] M. Lanza, “CodeCrawler - Polymetric Views in Action,” Proceedings of the *19th IEEE International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2004, pp. 394-395.

[97] M. Lanza and S. Ducasse, “Polymetric Views - A Lightweight Visual Approach to Reverse Engineering,” *IEEE Transactions on Software Engineering*, vol. 29, 2003, pp. 782-795.

[98] C. Larman and V.R. Basili, “Iterative and Incremental Development: A Brief History,” *IEEE Computer*, vol. 36, 2003, pp. 47-56.

[99] M.M. Lehman and F.N. Parr, “Program Evolution and Its Impact on Software Engineering,” *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, San Francisco, California, United States: IEEE Computer Society Press, 1976, pp. 350-357.

[100] W. Li and S. Henry, “Object-Oriented Metrics that Predict Maintainability,” *Journal of Systems and Software (JSS)*, vol. 23, 1993, pp. 111-122.

[101] J.M. Lions, D. Simoneau, G. Pitette, and I. Moussa, “Extending OpenTool/UML Using Metamodeling: An Aspect Oriented Programming Case Study,” *Proceedings of the 2nd Workshop on Aspect-Oriented Modeling with UML*, 2002.

[102] R. Lopez-Herrejon and S. Apel, “Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies,” *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2007, pp. 423-437.

[103] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall PTR, NY, 1994.

[104] A. Lozano and M. Wermelinger, “Assessing the Effect of Clones on Changeability,” *IEEE International Conference on Software Maintenance (ICSM)*, 2008, pp. 227-236.

[105] A. Lozano, M. Wermelinger, and B. Nuseibeh, “Assessing the Impact of Bad Smells using Historical Information,” *9th International Workshop on Principles of Software Evolution in conjunction with the 6th ESEC/FSE joint meeting*, Dubrovnik, Croatia: ACM, 2007, pp. 31-34.

[106] A. Marcus and J.I. Maletic, “Identification of High-Level Concept Clones in Source Code,” *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2001, p. 107.

[107] M. Marin, L. Moonen, and A.V. Deursen, “SoQueT: Query-Based Documentation of Crosscutting Concerns,” *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2007, pp. 758-761.

[108] M. Marin, L. Moonen, and A.V. Deursen, “An Approach to Aspect Refactoring based on Crosscutting Concern Types,” *Proceedings of the Workshop on Modeling and Analysis of Concerns in Software (MACS)*, St. Louis, Missouri: ACM, 2005, pp. 1-5.

[109] M. Marin, L. Moonen, and A.V. Deursen, “A Classification of Crosscutting Concerns,” *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2005, pp. 673-676.

[110] M. Marin, A.V. Deursen, and L. Moonen, “Identifying Aspects Using Fan-In Analysis,” *Proceedings of the 11th Working Conference on Reverse Engineering*, IEEE Computer Society, 2004, pp. 132-141.

[111] R. Marinescu, “Detection Strategies: Metrics-Based Rules for Detecting Design Flaws,” *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2004, pp. 350-359.

[112] R. Marinescu, “Measurement and Quality in Object-Oriented Design,” *PhD Thesis*, "Politehnica" University of Timisoara, 2002.

[113] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall PTR, 2000.

[114] M. Mezini and K. Ostermann, “Conquering Aspects with Caesar,” *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, Boston, Massachusetts: ACM, 2003, pp. 90-99.

[115] A. Mitchell and J.F. Power, “Toward a definition of run-time object-oriented metrics,” *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, Berlin: Springer, 2003.

[116] The MobileMedia Project. Accessed: 30 Sept 2009, URL http://sourceforge.net/projects/mobilemedia/

[117] M. Monteiro and J. Fernandes, "Towards a Catalogue of Refactorings and Code Smells for AspectJ," *Transactions on Aspect-Oriented Software Development I*, 2006, pp. 214-258.

[118] M. Monteiro and J. Fernandes, "Towards a Catalog of Aspect-Oriented Refactorings," *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, New York, NY, USA: ACM, 2005, pp. 111–122.

[119] L. Moonen, "Dealing with Crosscutting Concerns in Existing Software," *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM), Frontiers of Software Maintenance*, 2008, pp. 68-77.

[120] A. Oliveira, L. Cysneiros, J. Leite, E. Figueiredo, and C. Lucena, "Integrating Scenarios, i*, and AspectT in the Context of Multi-Agent Systems," *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada: ACM, 2006, p. 16.

[121] OMG Unified Modeling Language Specification. Accessed: 30 Sept 2009, URL http://www.omg.org/docs/formal/03-03-01.pdf

[122] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD Thesis, University of Illinois at Urbana-Champaign, 1992.

[123] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, 1972, pp. 1053-1058.

[124] K. Pohl, G. Böckle, and F.J.V.D. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, 2005.

[125] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Science/Engineering/Math, 2001.

[126] B. Ramesh and M. Jarke, "Toward Reference Models for Requirements Traceability," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, 2001, pp. 58-93.

[127] A.J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley Professional, 1996.

[128] D.B. Roberts, "Practical Analysis for Refactoring," PhD Thesis, University of Illinois at Urbana-Champaign, 1999.

[129] M.P. Robillard and G.C. Murphy, "Representing Concerns in Source Code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, 2007, p. 3.

[130] M.P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. *An Empirical Study of the Concept Assignment Problem*. Technical Report TR-2007.3, School of Computer Science, McGill University, 2007.

[131] H.A. Sahraoui, R. Godin, and T. Miceli, "Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and Its Automation?," *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2000, p. 154.

[132] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. Lucena, "On the Modularity Assessment of Software Architectures: Do my architectural concerns count?," *First Workshop on Aspects in Architectural Description (ARRCH)*, Vancouver, 2007.

[133] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. Lucena, "On the Modularity of Software Architectures: A Concern-Driven Measurement Framework," *Proceedings of the 1st European Conference on Software Architecture (ECSA)*, 2007, pp. 207-224.

[134] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. von Staa, "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework," *Proceedings of the XVII Brazilian Symposium on Software Engineering*, 2003.

[135] B. Silva, "A Refactoring Method for Crosscutting Concern Modularisation," MSc Dissertation in Computer Science, *in Portuguese*, Federal University of Rio Grande do Sul, 2009.

[136] B. Silva, E. Figueiredo, A. Garcia, and D. Nunes, “Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics,” *Electronic Notes Theoretical Computer Science*, vol. 233, 2009, pp. 105-125.

[137] S. Soares, E. Laureano, and P. Borba, “Implementing Distribution and Persistence Aspects with AspectJ,” *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, New York, NY, USA: ACM, 2002, pp. 174–190.

[138] I. Sommerville, *Software Engineering*, Addison Wesley, 2004.

[139] J.T. Stasko, J.B. Domingue, M.H. Brown, B.A. Price, and J. Foley, *Software Visualization*, The MIT Press, 1998.

[140] W. Stevens, G. Myers, and L. Constantine, “Structured Design,” *IBM Systems Journal*, vol. 13, 1974, pp. 139, 115.

[141] S. Stevens, “On the Theory of Scales of Measurement,” *Science*, vol. 103, Jun. 1946, pp. 680, 677.

[142] K.J. Sullivan, W.G. Griswold, Y. Cai, and B. Hallen, “The structure and value of modularity in software design,” *Proceedings of the 8th European Software Engineering Conference (ESEC)*, Vienna, Austria: ACM, 2001, pp. 99-108.

[143] S.M. Sutton and I. Rouvellou, “Modeling of Software Concerns in Cosmos,” *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, The Netherlands: ACM, 2002, pp. 127-133.

[144] C. Szyperski, *Component Software: Beyond Object-Oriented Programming* (2nd Edition), Addison-Wesley Professional, 2002.

[145] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton, “N Degrees of Separation: Multi-Dimensional Separation of Concerns,” *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, Los Angeles, California, United States: ACM, 1999, pp. 107-119.

[146] B. Tekinerdogan, “ASAAM: Aspectual Software Architecture Analysis Method,” *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE Computer Society, 2004, p. 5.

[147] S.S. Yau and J.S. Collofello, “Design Stability Measures for Software Maintenance,” *IEEE Transactions on Software Engineering (TSE)*, vol. 11, 1985, pp. 849-856.

[148] T. Young, “Using AspectJ to Build a Software Product Line for Mobile Devices,” MSc Dissertation in Computer Science, University of British Columbia, 2005.

[149] T. Young and G. Murphy. “Using AspectJ to Build a Product Line for Mobile Devices”. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, demo session, Chicago, 2005.

[150] M. Wermelinger, Yijun Yu, and A. Lozano, “Design Principles in Architectural Evolution: A Case Study,” *IEEE International Conference on Software Maintenance (ICSM)*, 2008, pp. 396-405.

[151] W.E. Wong, S.S. Gokhale, and J.R. Horgan, “Quantifying the Closeness between Program Components and Features,” *Journal of Systems and Software (JSS)*, vol. 54, 2000, pp. 87-98.

[152] J. Zhao, “Measuring Coupling in Aspect-Oriented Systems,” *Proceedings of the 10th International Software Metrics Symposium*, 2004.

[153] J. Zhao and B. Xu, “Measuring Aspect Cohesion,” *Proceeding International Conference on Fundamental Approaches to Software Engineering (FASE)*, Springer-Verlag, 2004, pp. 54-68.

[154] J. Zhao, *Towards a Metrics Suite for Aspect-Oriented Software*, Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), 2002.

[155] J. L. Myers and A. D. Well. *Research Design and Statistical Analysis* (2nd Edition), Lawrence Erlbaum, 2003.

Appendix A: Questionnaires given to the systems' specialists

Questionnaire
Design Patterns Library

1. Your name: *<confidential>*
2. Male ( ) Female ( )

3. Provide below your experience with respect to *the Design Patterns Library*.

   a. Choose which design(s) of the target system you are familiar with.

( ) OO design (Java) ( ) I am familiar with both AO and OO designs

( ) AO design (AspectJ) ( ) I am not familiar with either designs

   b. Tick which of the following activities you performed in this system.

[ ] I took part in discussions about its design.

[ ] I designed or implemented (part of) it.

[ ] I analysed its design quality attributes.

   c. How long have you worked with this system?

( ) 0 to 6 months ( ) 1 to 3 years

( ) 6 months to 1 year ( ) More than 3 years

4. If you are familiar with both OO and AO designs (Question 3.a), answer the following question.

   a. According to your expertise, which solution do you think is best to implement the following concerns in this system?
   You can leave blank if you are unsure about the best solution.

Director role (Builder)	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Handler role (Chain of Resp.)	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Creator role (Factory Method)	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Observer role (Observer)	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Subject role (Observer)	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Colleague role (Mediator)	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Mediator role (Mediator)	( ) OO / Java	( ) AO / AspectJ	( ) Equally good

Questionnaire
OpenOrb-complaint Middleware System

1. Your name: *<confidential>*
2. Male ( ) Female ( )

3. Provide below your experience with respect to *the OpenOrb-complaint Middleware System*.

   a. Choose which design(s) of the target system you are familiar with.

( ) OO design (Java)

( ) AO design (AspectJ)

( ) I am familiar with both AO and OO designs

( ) I am not familiar with either designs

   b. Tick which of the following activities you performed in this system.

[ ] I took part in discussions about its design.

[ ] I designed or implemented (part of) it.

[ ] I analysed its design quality attributes.

   c. How long have you worked with this system?

( ) 0 to 6 months

( ) 6 months to 1 year

( ) 1 to 3 years

( ) More than 3 years

4. If you are familiar with both OO and AO designs (Question 3.a), answer the following question.

   d. According to your expertise, which solution do you think is best to implement the following design patterns in this system?
   You can leave blank if you are unsure about the best solution.

Factory Method	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Observer	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Facade	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Singleton	( ) OO / Java	( ) AO / AspectJ	( ) Equally good

Questionnaire
Measurement Tool

1. Your name: *<confidential>*
2. Male ( ) Female ( )

3. Provide below your experience with respect to *the measurement tool*.

   a. Choose which design(s) of the target system you are familiar with.

( ) OO design (Java)

( ) AO design (AspectJ)

( ) I am familiar with both AO and OO designs

( ) I am not familiar with either designs

   b. Tick which of the following activities you performed in this system.

[ ] I took part in discussions about its design.

[ ] I designed or implemented (part of) it.

[ ] I analysed its design quality attributes.

   c. How long have you worked with this system?

( ) 0 to 6 months

( ) 6 months to 1 year

( ) 1 to 3 years

( ) More than 3 years

4. If you are familiar with both OO and AO designs (Question 3.a), answer the following question.

   d. According to your expertise, which solution do you think is best to implement the following design patterns in this system?
   You can leave blank if you are unsure about the best solution.

Prototype	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
State	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Interpreter	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Proxy	( ) OO / Java	( ) AO / AspectJ	( ) Equally good

Questionnaire
CVS Core Plugin

1. Your name: *<confidential>*
2. Male ( ) Female ( )

3. Provide below your experience with respect to *the CVS Core Plugin*.

   a. Choose which design(s) of the target system you are familiar with.

   ( ) OO design (Java)

   ( ) AO design (AspectJ)

   ( ) I am familiar with both AO and OO designs

   ( ) I am not familiar with either designs

   b. Tick which of the following activities you performed in this system.

   [ ] I took part in discussions about its design.

   [ ] I designed or implemented (part of) it.

   [ ] I analysed its design quality attributes.

   c. How long have you worked with this system?

   ( ) 0 to 6 months

   ( ) 6 months to 1 year

   ( ) 1 to 3 years

   ( ) More than 3 years

4. If you are familiar with both OO and AO designs (Question 3.a), answer the following question.

   d. According to your expertise, which solution do you think is best to implement *exception handling* in this system?
   You can leave blank if you are unsure about the best solution.

   ( ) OO / Java ( ) AO / AspectJ ( ) Equally good

Questionnaire
Health Watcher

1. Your name: *<confidential>*
2. Male ( ) Female ( )

3. Provide below your experience with respect to *Health Watcher*.

   a. Choose which design(s) of the target system you are familiar with.

( ) OO design (Java) ( ) I am familiar with both AO and OO designs

( ) AO design (AspectJ) ( ) I am not familiar with either designs

   b. Tick which of the following activities you performed in this system.

[ ] I took part in discussions about its design.

[ ] I designed or implemented (part of) it.

[ ] I analysed its design quality attributes.

   c. How long have you worked with this system?

( ) 0 to 6 months ( ) 1 to 3 years

( ) 6 months to 1 year ( ) More than 3 years

4. If you are familiar with both OO and AO designs (Question 3.a), answer the following question.

   d. According to your expertise, which solution do you think is best to implement the following concerns in this system?
   You can leave blank if you are unsure about the best solution.

Business	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Concurrency	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Distribution	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Persistence	( ) OO / Java	( ) AO / AspectJ	( ) Equally good

Questionnaire
Portalware

1. Your name: *<confidential>*
2. Male ( ) Female ( )

3. Provide below your experience with respect to *Portalware*.

   a. Choose which design(s) of the target system you are familiar with.

( ) OO design (Java)

( ) AO design (AspectJ)

( ) I am familiar with both AO and OO designs

( ) I am not familiar with either designs

   b. Tick which of the following activities you performed in this system.

[ ] I took part in discussions about its design.

[ ] I designed or implemented (part of) it.

[ ] I analysed its design quality attributes.

   c. How long have you worked with this system?

( ) 0 to 6 months

( ) 6 months to 1 year

( ) 1 to 3 years

( ) More than 3 years

4. If you are familiar with both OO and AO designs (Question 3.a), answer the following question.

   d. According to your expertise, which solution do you think is best to implement the following concerns in this system?
   You can leave blank if you are unsure about the best solution.

Adaptation	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Autonomy	( ) OO / Java	( ) AO / AspectJ	( ) Equally good
Collaboration	( ) OO / Java	( ) AO / AspectJ	( ) Equally good

Appendix B: Architectural Crosscutting Pattern Instances

Table B-1: Components forming Black Sheep in the MobileMedia architecture

Concerns	R1	R2	R3	R4	R5	R6	R7
Album	0	0	0	0	0	0	0
Capture	-	-	-	-	-	-	0
Controller	0	0	0	0	0	0	0
Copy	-	-	-	0	0	4	6
ExceptionHandling	0	0	0	0	0	0	0
Favourites	-	-	3	3	3	3	3
Labelling	3	0	0	0	0	0	0
Media	-	-	-	-	-	0	0
Music	-	-	-	-	-	0	0
Persistence	0	0	0	0	0	0	0
Photo	0	0	0	0	0	0	0
SMS	-	-	-	-	0	0	0
Sorting	-	3	3	3	3	3	3
Video	-	-	-	-	-	-	0

Table B-2: Components forming Octopus in the MobileMedia architecture

Concerns	R1	R2	R3	R4	R5	R6	R7
Album	0	4	4	10	10	12	15
Capture	-	-	-	-	-	-	6
Controller	0	0	0	0	0	0	0
Copy	-	-	-	0	0	0	0
ExceptionHandling	3	3	3	6	7	10	14
Favourites	-	-	0	0	0	0	0
Labelling	0	4	4	8	8	10	13
Media	-	-	-	-	-	10	12
Music	-	-	-	-	-	6	6
Persistence	3	3	3	5	6	9	13
Photo	8	9	9	10	11	9	9
SMS	-	-	-	-	3	3	4
Sorting	-	0	0	0	0	0	0
Video	-	-	-	-	-	-	8

Table B-3: Components forming King Snake in the MobileMedia architecture

Concerns	R1	R2	R3	R4	R5	R6	R7
Album	3	4	4	6	6	6	6
Capture	-	-	-	-	-	-	4
Controller	0	2	2	5	5	5	5
Copy	-	-	-	2	2	2	2
ExceptionHandling	3	3	3	3	3	3	3
Favourites	-	-	2	2	2	2	2
Labelling	2	4	4	4	4	4	4
Media	-	-	-	-	-	7	7
Music	-	-	-	-	-	5	5
Persistence	3	3	3	3	3	3	4
Photo	3	3	3	6	6	5	4
SMS	-	-	-	-	4	4	4
Sorting	-	2	2	2	2	2	2
Video	-	-	-	-	-	-	4

Table B-4: Components forming Tsunami in the MobileMedia architecture

Concerns	R1	R2	R3	R4	R5	R6	R7
Album	4	4	5	6	6	8	13
Capture	-	-	-	-	-	-	3
Controller	0	0	0	5	5	5	5
Copy	-	-	-	0	0	0	0
ExceptionHandling	3	3	3	3	3	5	7
Favourites	-	-	0	0	0	0	0
Labelling	3	5	5	5	5	6	7
Media	-	-	-	-	-	11	13
Music	-	-	-	-	-	5	5
Persistence	3	3	3	3	3	5	7
Photo	6	6	6	8	8	7	7
SMS	-	-	-	-	4	4	4
Sorting	-	0	0	0	0	0	0
Video	-	-	-	-	-	-	4

Table B-5: Components forming Tree Root in the MobileMedia architecture

Concerns	R1	R2	R3	R4	R5	R6	R7
Album	4	4	5	9	10	11	13
Capture	-	-	-	-	-	-	5
Controller	0	0	0	5	6	7	8
Copy	-	-	-	0	0	0	0
ExceptionHandling	3	3	3	6	6	7	7
Favourites	-	-	3	3	3	3	3
Labelling	0	4	4	6	6	6	6
Media	-	-	-	-	-	12	14
Music	-	-	-	-	-	6	6
Persistence	3	3	3	5	6	6	6
Photo	8	9	9	12	13	7	7
SMS	-	-	-	-	4	4	4
Sorting	-	3	3	3	3	3	3
Video	-	-	-	-	-	-	7

Appendix C: Crosscutting Pattern Instances

Table C-1: Crosscutting patterns instances in the Health Watcher design (Java)

Concerns	BS	Oct	GC	CP	HD	TR	Tsu
Concurrency		1				1	1
Distribution		1			1	1	1
Persistence		1	1			1	1
Abstract Factory (1)				1		1	
Abstract Factory (2)		1		1			
Adapter (1)				1		1	1
Adapter (2)	1						
Command			1	1		1	1
Observer		1		1	1	1	
State				1		1	1

Table C-1 (b): Crosscutting patterns instances in the Health Watcher design (Java)

Concerns	KS	NN	CC	DS	DC	BC	Total
Concurrency	1		1				**5**
Distribution		1	1				**6**
Persistence		1	1				**6**
Abstract Factory (1)	1	1					**4**
Abstract Factory (2)	1	1					**4**
Adapter (1)	1						**4**
Adapter (2)							**1**
Command			1				**5**
Observer			1				**5**
State			1				**4**

Key of crosscutting patterns: BS - Black Sheep; Oct - Octopus; GC - God Concern; CP - Climbing Plant; HD - Hereditary Disease; TR - Tree Root; Tsu - Tsunami; KS - King Snake; NN - Neural Network; CC - Copy Cat; DS - Dolly Sheep; DC - Data Concern; BC - Behavioral Concern

Table C-2: Crosscutting patterns instances in the Health Watcher design (AspectJ)

Concerns	**BS**	**Oct**	**GC**	**CP**	**HD**	**TR**	**Tsu**
Concurrency							
Distribution						1	1
Persistence							
Abstract Factory (1)				1		1	
Abstract Factory (2)							
Adapter (1)							
Adapter (2)							
Command							
Observer							
State				1		1	1

Table C-2 (b): Crosscutting patterns instances in the Health Watcher design (AspectJ)

Concerns	**KS**	**NN**	**CC**	**DS**	**DC**	**BC**	**Total**
Concurrency							**0**
Distribution		1	1				**4**
Persistence							**0**
Abstract Factory (1)							**2**
Abstract Factory (2)							**0**
Adapter (1)							**0**
Adapter (2)							**0**
Command							**0**
Observer							**0**
State			1				**4**

Table C-3: Crosscutting patterns instances in the Design Pattern library (Java)

Concerns	BS	Oct	GC	CP	HD	TR	Tsu
Factory (Ab. Factory)			1	1	1		1
Product (Ab. Factory)	1						
Adapter (Adapter)					1		
Adaptee (Adapter)							
Target (Adapter)				1	1		
Abstraction (Bridge)				1			1
Implementor (Bridge)			1	1			1
Builder (Builder)			1	1			1
Director (Builder)	1						
Handler (Chain of R.)				1		1	
Command (Comm.)			1	1			
Invoker (Command)							
Receiver (Command)					1	1	
Composite (Comp.)							1
Component (Comp.)				1		1	
Leaf (Composite)							
Decorator (Decorator)				1		1	
Component (Decor.)		1	1	1			
Façade (Façade)		1					
Product (Fac. Method)	1			1	1		
Creator (Fac. Method)			1	1			1
Flyweight (Flyweight)				1			
Factory (Flyweight)							
Context (Interpreter)							
Expression (Interp.)		1		1			
Aggregate (Iterator)				1		1	1
Iterator (Iterator)		1					
Colleague (Mediator)		1		1			
Mediator (Mediator)		1		1			
Memento (Memento)							
Originator (Memento)							1
Subject (Observer)		1		1			1
Observer (Observer)		1		1			
Prototype (Prototype)	1						1
Proxy (Proxy)			1		1		1
Singleton (Singleton)	1				1		1
Context (State)				1			
State (State)			1	1			
Context (Strategy)							1
Strategy (Strategy)			1	1			1
Abstract Class (TM)		1		1			
Concrete Class (TM)			1		1		1
Element (Visitor)		1		1			
Visitor (Visitor)		1		1	1	1	1

Table C-4: Crosscutting patterns instances in the Design Pattern library (Java)

Concerns	KS	NN	CC	DS	DC	BC	Total
Factory (Ab. Factory)						1	**5**
Product (Ab. Factory)			1	1			**3**
Adapter (Adapter)	1						**2**
Adaptee (Adapter)		1					**1**
Target (Adapter)	1						**3**
Abstraction (Bridge)							**2**
Implementor (Bridge)			1			1	**5**
Builder (Builder)							**3**
Director (Builder)							**1**
Handler (Chain of R.)			1				**3**
Command (Comm.)		1					**3**
Invoker (Command)							**0**
Receiver (Command)							**2**
Composite (Comp.)							**1**
Component (Comp.)						1	**3**
Leaf (Composite)						1	**1**
Decorator (Decorator)							**2**
Component (Decor.)							**3**
Façade (Façade)							**1**
Product (Fac. Method)							**3**
Creator (Fac. Method)							**3**
Flyweight (Flyweight)	1		1				**3**
Factory (Flyweight)	1						**1**
Context (Interpreter)	1						**1**
Expression (Interp.)			1				**3**
Aggregate (Iterator)							**3**
Iterator (Iterator)	1						**2**
Colleague (Mediator)							**2**
Mediator (Mediator)						1	**3**
Memento (Memento)		1					**1**
Originator (Memento)							**1**
Subject (Observer)			1				**4**
Observer (Observer)						1	**3**
Prototype (Prototype)			1	1		1	**5**
Proxy (Proxy)			1				**4**
Singleton (Singleton)							**3**
Context (State)	1						**2**
State (State)		1	1				**4**
Context (Strategy)					1		**2**
Strategy (Strategy)			1			1	**5**
Abstract Class (TM)	1						**3**
Concrete Class (TM)						1	**4**
Element (Visitor)						1	**3**
Visitor (Visitor)			1				**6**

Table C-5: Crosscutting patterns instances in the Design Pattern library (AspectJ)

Concerns	BS	Oct	GC	CP	HD	TR	Tsu
Factory (Ab. Factory)			1				1
Product (Ab. Factory)	1						
Adapter (Adapter)							
Adaptee (Adapter)						1	
Target (Adapter)						1	
Abstraction (Bridge)							1
Implementor (Bridge)				1			1
Builder (Builder)				1			1
Director (Builder)							
Handler (Chain of R.)							
Command (Comm.)						1	
Invoker (Command)							
Receiver (Command)	1						
Composite (Comp.)							
Component (Comp.)		1					
Leaf (Composite)	1						
Decorator (Decorator)							
Component (Decor.)							
Façade (Façade)		1					
Product (Fac. Method)							
Creator (Fac. Method)			1	1			1
Flyweight (Flyweight)		1		1			
Factory (Flyweight)							
Context (Interpreter)							
Expression (Interp.)							
Aggregate (Iterator)						1	
Iterator (Iterator)							
Colleague (Mediator)		1					
Mediator (Mediator)		1					
Memento (Memento)						1	
Originator (Memento)							
Subject (Observer)		1				1	
Observer (Observer)							
Prototype (Prototype)							
Proxy (Proxy)							
Singleton (Singleton)							
Context (State)							
State (State)							
Context (Strategy)							
Strategy (Strategy)		1					1
Abstract Class (TM)		1		1		1	
Concrete Class (TM)							
Element (Visitor)							
Visitor (Visitor)							

Table C-6: Crosscutting patterns instances in the Design Pattern library (AspectJ)

Concerns	KS	NN	CC	DS	DC	BC	Total
Factory (Ab. Factory)							**2**
Product (Ab. Factory)			1	1			**3**
Adapter (Adapter)							**0**
Adaptee (Adapter)							**1**
Target (Adapter)							**1**
Abstraction (Bridge)							**1**
Implementor (Bridge)						1	**3**
Builder (Builder)							**2**
Director (Builder)							**0**
Handler (Chain of R.)	1						**1**
Command (Comm.)		1					**2**
Invoker (Command)							**0**
Receiver (Command)							**1**
Composite (Comp.)							**0**
Component (Comp.)						1	**2**
Leaf (Composite)							**1**
Decorator (Decorator)							**0**
Component (Decor.)							**0**
Façade (Façade)							**1**
Product (Fac. Method)							**0**
Creator (Fac. Method)							**3**
Flyweight (Flyweight)	1						**3**
Factory (Flyweight)	1						**1**
Context (Interpreter)	1						**1**
Expression (Interp.)							**0**
Aggregate (Iterator)							**1**
Iterator (Iterator)							**0**
Colleague (Mediator)							**1**
Mediator (Mediator)						1	**2**
Memento (Memento)							**1**
Originator (Memento)							**0**
Subject (Observer)							**2**
Observer (Observer)						1	**1**
Prototype (Prototype)							**0**
Proxy (Proxy)							**0**
Singleton (Singleton)							**0**
Context (State)							**0**
State (State)							**0**
Context (Strategy)							**0**
Strategy (Strategy)						1	**3**
Abstract Class (TM)							**3**
Concrete Class (TM)							**0**
Element (Visitor)							**0**
Visitor (Visitor)							**0**

Appendix D: Crosscutting Patterns per Architetural Component

Table D-1: Unstable architectural components and crosscutting patterns (Release 1).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	KS	Tsu	TR		
AddPhotoToAlbumScreen	0	3	1	1	1	6	5
AlbumData	1	6	3	5	4	19	4
AlbumListScreen	0	3	2	2	2	9	2
BaseController	1	7	5	5	4	22	5
ImageAcessor	0	5	2	3	4	14	3
NewAlbumScreen	0	3	1	1	1	6	3
PhotoListScreen	1	3	0	2	1	7	4
PhotoViewScreen	0	2	0	0	1	3	3
Average	**0.4**	**4.0**	**1.8**	**2.4**	**2.3**	**10.8**	**3.6**

Table D-2: Unstable architectural components and crosscutting patterns (Release 2).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	KS	Tsu	TR		
AddPhotoToAlbumScreen	0	2	1	1	1	5	4
AlbumData	0	2	3	5	4	14	3
AlbumListScreen	0	2	2	2	2	8	2
BaseController	1	3	7	5	6	22	4
ImageAcessor	0	1	3	3	4	11	2
NewLabelScreen	0	3	1	2	2	8	3
PhotoController	1	1	3	1	3	9	4
PhotoListScreen	1	2	1	2	3	9	4
PhotoViewScreen	0	1	0	0	1	2	3
Average	**0.3**	**1.9**	**2.3**	**2.3**	**2.9**	**9.8**	**3.2**

Table D-3: Unstable architectural components and crosscutting patterns (Release 3).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	KS	Tsu	TR		
AddPhotoToAlbumScreen	0	3	1	1	1	6	4
AlbumData	0	10	3	5	4	22	2
AlbumListScreen	0	4	2	2	2	10	2
BaseController	2	13	8	5	7	35	3
ImageAcessor	0	7	3	4	4	18	2
NewLabelScreen	0	6	1	2	3	12	2
PhotoController	2	6	3	1	4	16	4
PhotoListScreen	2	6	2	2	4	16	3
PhotoViewScreen	0	2	0	0	1	3	3
Average	**0.7**	**6.3**	**2.6**	**2.4**	**3.3**	**15.3**	**2.8**

Table D-4: Unstable architectural components and crosscutting patterns (Release 4).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	KS	Tsu	TR		
AddPhotoToAlbumScreen	0	3	0	1	1	5	4
AlbumController	0	5	3	3	6	17	1
AlbumData	0	5	5	5	5	20	2
AlbumListScreen	0	2	2	2	2	8	2
BaseController	0	2	1	1	3	7	2
ImageAcessor	0	4	4	4	4	16	2
NewLabelScreen	0	3	2	2	3	10	2
PhotoController	2	4	7	5	8	26	3
PhotoListController	0	4	2	4	5	15	1
PhotoListScreen	2	1	4	2	4	13	2
PhotoViewController	2	6	2	1	7	18	3
PhotoViewScreen	0	2	1	0	1	4	3
Average	**0.5**	**3.4**	**2.7**	**2.5**	**4**	**13.2**	**2.3**

Table D-5: Unstable architectural components and crosscutting patterns (Release 5).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	KS	Tsu	TR		
AddPhotoToAlbumScr.	0	3	0	1	1	5	3
AlbumController	0	5	2	3	6	16	1
AlbumData	0	5	5	5	5	20	1
AlbumListScreen	0	2	2	1	2	7	1
BaseController	0	2	1	2	3	8	1
ImageAcessor	0	4	4	4	4	16	1
NetworkScreen	0	0	1	1	1	3	0
NewLabelScreen	0	3	1	1	3	8	1
PhotoController	2	4	7	5	8	26	2
PhotoListController	0	3	4	3	4	14	1
PhotoListScreen	2	1	3	1	3	10	2
PhotoViewController	2	5	4	3	8	22	3
PhotoViewScreen	0	1	2	2	3	8	2
SMSController	0	4	1	2	6	13	2
Average	**0.4**	**3.0**	**2.6**	**2.4**	**4.1**	**12.6**	**1.5**

Table D-6: Unstable architectural components and crosscutting patterns (Release 6).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	KS	Tsu	TR		
AddMediaToAlbumScr.	0	4	0	2	1	7	2
AlbumController	0	6	4	3	7	20	1
AlbumListScreen	0	2	2	2	2	8	1
AlbumMusicData	0	5	1	6	1	13	0
AlbumPhotoData	0	5	6	6	6	23	1
BaseController	0	2	1	2	3	8	1
ImageAcessor	0	4	5	5	5	19	1
MediaController	2	7	8	7	10	34	2
MediaListController	0	4	3	4	5	16	1
MediaListScreen	2	4	3	2	6	17	2
MusicAcessor	0	4	1	5	1	11	0
NetworkScreen	0	0	1	1	1	3	0
NewLabelScreen	0	2	2	1	2	7	1
PhotoViewController	3	5	4	3	9	24	2
PhotoViewScreen	1	1	3	2	2	9	1
PlayMusicController	1	5	1	2	4	13	0
PlayMusicScreen	1	1	1	1	1	5	0
SMSController	0	5	2	2	6	15	2
Average	**0.6**	**3.7**	**2.7**	**3.1**	**4.0**	**14.0**	**1.0**

Table D-7: Unstable architectural components and crosscutting patterns (Release 7).

Architectural Elements	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	KS	Tsu	TR		
AddMediaToAlbumScr.	0	4	1	2	3	10	1
AlbumController	0	6	3	3	6	18	0
AlbumListScreen	0	2	2	2	2	8	0
AlbumMusicData	0	5	2	6	2	15	0
AlbumPhotoData	0	5	4	6	5	20	0
AlbumVideoData	0	5	1	6	1	13	0
BaseController	0	2	1	3	3	9	0
CaptureMediaController	0	7	2	3	4	16	0
CaptureMediaScreen	0	3	2	3	1	9	0
ImageAcessor	0	4	4	5	4	17	0
MediaController	2	9	8	7	11	37	1
MediaListController	0	4	4	5	4	17	0
MediaListScreen	2	6	5	3	9	25	1
MusicAcessor	0	4	2	5	2	13	0
NetworkScreen	0	0	1	1	1	3	0
NewLabelScreen	0	2	2	1	3	8	0
PhotoViewController	3	6	5	4	8	26	1
PhotoViewScreen	1	1	2	2	1	7	1
PlayMusicController	1	5	2	1	5	14	0
PlayMusicScreen	1	1	1	1	1	5	0
PlayVideoController	1	6	0	0	4	11	0
PlayVideoScreen	1	1	0	0	1	3	0
SMSController	0	4	1	1	6	12	1
VideoAcessor	0	4	1	5	2	12	0
Average	**0.6**	**3.7**	**2.7**	**3.1**	**4.0**	**14.0**	**1.0**

Appendix E: Crosscutting Patterns per Component

Table E-1: Crosscutting patterns in the object-oriented design of MobileMedia (R2)

Components	BS	Oct	GC	CP	HD	KS	TR	Tsu	NN	BC	DC	Sum
AddPhotoToAlbum	0	2	1	0	0	0	0	0	0	0	0	**3**
AlbumData	0	4	3	0	0	2	3	3	3	0	0	**18**
AlbumListScreen	0	1	1	0	0	0	0	2	2	0	0	**6**
BaseController	1	4	3	0	0	1	2	3	3	1	0	**18**
BaseThread	0	0	0	0	0	0	0	0	0	0	0	**0**
Constants	0	0	1	0	0	0	0	0	0	0	0	**1**
ControllerInterface	0	1	0	0	0	0	0	0	0	0	0	**1**
ImageAccessor	0	3	3	0	0	2	5	3	5	0	0	**21**
ImageData	1	2	2	0	0	1	2	4	4	1	0	**17**
ImageNotFoundException	0	2	3	0	0	1	0	2	2	0	0	**10**
ImagePathNotValidExc.	0	2	3	3	0	2	3	1	3	0	0	**17**
ImageUtil	1	4	3	0	0	2	5	1	5	1	0	**22**
InvalidArrayFormatExc.	0	2	3	3	0	1	3	0	3	0	0	**15**
InvalidImageDataExc.	0	2	3	3	0	3	3	3	3	0	0	**20**
InvalidImageFormatExc.	0	2	3	3	0	2	3	1	3	0	0	**17**
InvalidPhotoAlbumName	0	3	3	0	0	2	0	3	3	0	0	**14**
MainUIMidlet	0	0	0	0	0	0	2	0	2	0	0	**4**
NewLabelScreen	0	2	0	0	0	0	0	1	1	0	0	**4**
NullAlbumDataReference	0	3	2	0	0	0	0	2	2	0	0	**9**
PersistenceMechanismExc	0	2	2	0	0	0	0	1	2	0	0	**7**
PhotoController	1	2	1	0	0	1	2	2	2	1	0	**12**
PhotoListScreen	1	0	1	0	0	0	0	0	0	1	0	**3**
PhotoViewScreen	0	3	3	0	0	1	2	0	3	0	0	**12**
SplashScreen	0	0	0	0	0	0	0	0	0	0	0	**0**
UnavailablePhotoAlbumE	0	2	2	0	0	0	0	2	2	0	0	**8**
Average	**0.2**	**1.9**	**1.8**	**0.5**	**0.0**	**0.8**	**1.4**	**1.4**	**2.1**	**0.2**	**0.0**	**10.4**

Key of crosscutting patterns:

BS - Black Sheep; Oct - Octopus; GC - God Concern; CP - Climbing Plant; HD - Hereditary Disease; KS - King Snake; TR - Tree Root; Tsu - Tsunami; NN - Neural Network; CC - Copy Cat; DS - Dolly Sheep; DC - Data Concern; BC - Behavioral Concern

Table E-2: Crosscutting patterns in the object-oriented design of MobileMedia (R4)

Components	BS	Oct	GC	CP	HD	TR	Tsu	KS	CC	DS	DC	BC	Sum
AbstractController		1		1		1	1	1					**5**
AddPhotoToAlbum		2	1				2	1					**6**
AlbumController		4	1	1		2	2	1	1				**12**
AlbumData		5	1			4	4	4	2				**20**
AlbumListScreen		1	1				1						**3**
BaseController		2	1	1		2	1	1					**8**
BaseMessaging													**0**
BaseThread													**0**
Constants		1	1				1						**3**
ControllerInterface		1		1		1	1	1					**5**
ImageAccessor		5	1			1	2	3	2				**14**
ImageData	2	3	1			2	2	2					**12**
ImageNotFoundExc.		3	1				2						**6**
ImagePathNotValid		3	1	3		3	2	2					**14**
ImageUtil	2	4	1			2	1	2					**12**
InvalidArrayFormat		3	1	2		3	2	2					**13**
InvalidImageData		3	1	3		3	3	3					**16**
InvalidImageFormat		3	1	3		2	2						**11**
InvalidPhotoAlbum		3	1				2						**6**
MainUIMidlet		2	1			1	1	1					**6**
NullAlbumDataRef.		2					2						**4**
NewLabelScreen		1					1						**2**
PersistenceMechExc.		2					2						**4**
PhotoController	2	6	1	1		6	2	5	2				**25**
PhotoListController	2	4	1	1		4	2	2	1				**17**
PhotoListScreen	2	2	1				1						**6**
PhotoViewController	2	6	1	1		7	2	6					**25**
PhotoViewScreen		5	1			1	1		1				**9**
ScreenSingleton													**0**
SplashScreen													**0**
UnavailablePhotoA.		2	1				2						**5**
Average	**0.4**	**2.5**	**0.7**	**0.6**	**0**	**1.4**	**1.5**	**1.2**	**0.3**	**0**	**0**	**0**	**8.7**

Table E-3: Crosscutting patterns in the object-oriented design of MobileMedia (R6)

Components	BS	Oct	GC	CP	HD	KS	TR	Tsu	NN	BC	DC	Sum
AbstractController	0	1	0	2	0	2	2	2	2	1	0	**12**
AddMediaToAlbum	0	6	1	0	0	0	0	0	0	0	0	**7**
AlbumController	0	4	2	2	0	0	2	0	2	1	0	**13**
AlbumData	0	6	1	2	3	5	6	5	7	0	0	**35**
AlbumListScreen	0	4	1	0	0	0	0	2	2	0	0	**9**
BaseController	0	4	1	2	0	2	3	2	5	1	0	**20**
BaseMessaging	0	1	0	1	0	0	0	0	0	0	0	**2**
BaseThread	0	0	0	0	0	0	0	0	0	0	0	**0**
Constants	0	0	0	0	0	0	0	0	0	0	0	**0**
ControllerInterface	0	1	0	1	0	1	1	1	1	1	0	**7**
ImageAlbumData	0	3	2	2	1	0	3	0	3	0	0	**14**
ImageMediaAccessor	0	5	1	2	0	0	2	0	2	0	0	**12**
ImageNotFoundException	0	3	2	0	0	1	0	3	3	0	0	**12**
ImagePathNotValidExc.	0	3	2	3	0	2	3	1	3	0	0	**17**
InvalidArrayFormatExc.	0	3	2	3	0	2	3	1	3	0	0	**17**
InvalidImageDataExc.	0	3	2	3	0	3	3	3	3	0	0	**20**
InvalidImageFormatExc.	0	3	2	3	0	2	2	1	2	0	0	**15**
InvalidPhotoAlbumName	0	3	1	0	0	2	0	3	3	0	0	**12**
MainUIMidlet	0	3	1	0	0	4	3	5	5	1	0	**22**
MediaAccessor	0	3	2	2	1	3	4	3	4	0	0	**22**
MediaController	2	10	2	2	0	2	3	2	3	3	0	**29**
MediaData	2	4	2	2	0	2	1	3	3	2	0	**21**
MediaListController	2	7	1	2	0	2	3	2	3	3	0	**25**
MediaListScreen	2	1	1	0	0	0	0	0	0	2	0	**6**
MediaUtil	2	5	2	1	0	1	4	0	4	2	0	**21**
MultiMediaData	2	4	2	2	0	0	1	0	2	2	0	**15**
MusicAlbumData	0	3	2	2	0	0	2	0	2	0	0	**11**
MusicMediaAccessor	0	5	2	2	0	0	2	0	2	0	0	**13**
MusicMediaUtil	0	4	2	1	0	0	1	0	1	0	0	**9**
MusicPlayController	0	5	1	2	0	1	3	1	3	1	0	**17**
NetworkScreen	0	1	0	0	0	0	0	0	1	0	0	**2**
NewLabelScreen	0	2	0	0	0	0	0	0	0	0	0	**2**
PersistenceMechanismExc	0	2	1	0	0	1	0	2	2	0	0	**8**
PhotoViewController	0	4	2	2	0	0	4	0	4	1	0	**17**
PhotoViewScreen	0	3	0	0	0	0	2	0	2	0	0	**7**
PlayMediaScreen	0	3	0	0	0	1	1	1	1	0	0	**7**
ScreenSingleton	0	4	1	0	0	0	0	0	0	0	0	**5**
SelectMediaController	0	3	1	2	0	1	5	1	5	1	0	**19**
SelectTypeOfMedia	0	3	1	0	0	0	0	0	0	0	0	**4**
SmsMessaging	0	3	0	1	0	0	0	0	0	0	0	**4**
SmsReceiverController	0	4	0	2	0	2	3	2	3	1	0	**17**
SmsReceiverThread	0	2	0	0	0	2	2	2	2	0	0	**10**
SmsSenderController	0	5	2	2	0	0	4	0	4	1	0	**18**
SmsSenderThread	0	2	0	0	0	0	0	0	0	0	0	**2**
SplashScreen	0	0	0	0	0	0	0	0	0	0	0	**0**
UnavailablePhotoAlbumE	0	2	0	0	0	0	0	1	2	0	0	**5**
Average	**0.3**	**3.3**	**1.0**	**1.2**	**0.1**	**1.0**	**1.7**	**1.1**	**2.2**	**0.5**	**0.0**	**12.2**

Table E-4: Changes in the components of MobileMedia

Components	R1	R2	R3	R4	R5	R6	R7	# Changes
AbstractController			a	1		1	1	**2**
AddPhotoToAlbum				1	1	1		**3**
AlbumController			a	1		1	1	**2**
AlbumData	1	1				1	1	**3**
AlbumListScreen						1		**1**
BaseController	1	1	1	1		1	1	**5**
BaseMessaging				a	1			**1**
BaseThread								**0**
Constants		1						**1**
ControllerInterface				1				**1**
ImageAccessor	1	1		1	1	1		**5**
ImageData		1	1			1		**3**
ImageAlbumData					a	1		**1**
ImageMediaAccessor					a	1		**1**
ImageNotFoundException	1						1	**1**
ImagePathNotValidException	1						1	**1**
ImageUtil	1	1	1	1		1	1	**5**
InvalidArrayFormatException	1							**1**
InvalidImageDataException	1						1	**1**
InvalidImageFormatException	1						1	**1**
InvalidPhotoAlbumNameExc.	1						1	**1**
MainUIMidlet				1	1	1	1	**3**
NetworkScreen				a	1			**1**
NullAlbumDataReference	1							**1**
NewLabelScreen		1						**1**
PersistenceMechanismExc.	1							**1**
MultiMediaData					a	1		**1**
MusicAlbumData					a	1		**1**
MusicMediaAccessor					a	1		**1**
MusicMediaUtil					a	1		**1**
MusicPlayController					a	1		**1**
PhotoController	a	1	1	1	1	1		**5**
PhotoListController			a	1		1		**2**
PhotoListScreen		1	1			1		**3**
PhotoViewController			a	1	1	1	1	**3**
PhotoViewScreen	1			1	1	1		**4**
PlayMediaScreen					a	1		**1**
ScreenSingleton			a	1		1		**2**
SelectMediaController					a	1		**1**
SelectTypeOfMedia					a	1		**1**
SmsMessaging				a	1	1		**2**
SmsReceiverController				a	1	1		**2**
SmsReceiverThread				a	1			**1**
SmsSenderController				a	1	1		**2**
SmsSenderThread				a	1			**1**
SplashScreen								**0**
UnavailablePhotoAlbumExc.	1							**1**
Total	**14**	**9**	**5**	**13**	**13**	**29**	**12**	**83**

(a) Component added.

Table E-5: Crosscutting patterns in the object-oriented design of Health Watcher (R9)

Components	BS	Oct	GC	HD	TR	Tsu	KS	NN	CC	DS	DC	BC	Sum
AbstractFacadeFact.			1										**1**
AbstractRepositoryF.													**0**
Address		1	1						1				**3**
AddressRep.RDB		2			1	1		2	2				**8**
AnimalComplaint			1										**1**
AnimalCompl’State		1	1						1				**3**
AnimalC’StateClosed			1										**1**
AnimalC’StateOpen			1										**1**
ArrayRepositoryFact.													**0**
Command													**0**
CommandRequest													**0**
CommandResponse													**0**
CommunicationExc.		2			1	1		2					**6**
Complaint		4	1		1			1	3				**10**
ComplaintRecord		4	1		3			2					**10**
ComplaintRep’Array		3			1			1	1				**6**
ComplaintRep’RDB		3			3	1		2	2				**11**
ComplaintState		2	1						1				**4**
ComplaintStateClose			1										**1**
ComplaintStateOpen			1										**1**
ConcreteIterator													**0**
ConcurrencyManager		2			1		1	1	1				**6**
ConfigRMI		2			1	1		2	1				**7**
ConnectionPers’Exc.													**0**
Constants		2											**2**
Date		1			1			1	1				**4**
DiseaseRecord		2	1		1			1					**5**
DiseaseType		1	1						1				**3**
DiseaseTypeR’Array		2			1			1					**4**
DiseaseTypeR’RDB		2			2	1		2	2				**9**
Employee		3	1		1			1	3				**9**
EmployeeRecord		4	1		2		1	2					**10**
EmployeeR’Array		3			1			1	1				**6**
EmployeeR’RDB		2			2	1		2	2				**9**
ExceptionMessages													**0**
FacadeFactory													**0**
FacadeUnavailableE.													**0**
FoodComplaint			1										**1**
FoodComplaintState		1	1						1				**3**
FoodC’StateClosed			1										**1**
FoodC’StateOpen			1										**1**
Functions		1			1			1					**3**
GetData’DiseaseType		2	2		2			2	2				**10**
GetData’HealthUnit		2	2		2			2	2				**10**
GetData’Speciality		2	2		2			2	2				**10**
HealthUnit		3	1	1	1			1	3				**10**
HealthUnitRecord		3	1		1			1					**6**
HealthUnitRep’Array		2			1			1					**4**
HealthUnitRep’RDB		2			1	1		2	2				**8**
HealthWatcherFacade		4	1		3			2	2				**12**

continue in the next page…

Components	BS	Oct	GC	HD	TR	Tsu	KS	NN	CC	DS	DC	BC	Sum
HealthWatcherF’Init													**0**
HTMLCode			1										**1**
HWServer		1	1		1			1	1				**5**
HWServlet		1	2		1			1	1				**6**
IAddressRepository		3			1			1					**5**
IComplaintRepository		3			2			2					**7**
IDiseaseRepository		3			2			2					**7**
IEmployeeRepository		3			2			2					**7**
IFacade		3	1		3			3					**10**
IFacadeRMI’Adapter		2	1										**3**
IHealthUnitRep.		3			2			2					**7**
IIteratorRMI’Adapter		2			2			2					**6**
InsertAnimalCompl.		2	2		2			2	2				**10**
InsertDiseaseType		2	2		1			1	2				**8**
InsertEmployee		2	2		2			2	2				**10**
InsertEntryException		1			1			1					**3**
InsertFoodComplaint		2	2		2			2	2				**10**
InsertHealthUnit		2	2		1			1	2				**8**
InsertMedicalSpec.		2	2		1			1	2				**8**
InsertSpecialCompl.		2	2		2			2	2				**10**
InsertSymptom		2	2		1			1	2				**8**
InvalidDateException		1			1			1					**3**
InvalidSessionExc.		1			1			1					**3**
IPersistenceMech.		2			1	1		2					**6**
ISpecialityRepository		3			2			2					**7**
ISymptomRepository		3			2			2					**7**
IteratorDsk		2			1			1					**4**
IteratorRMIS’Adapt.		2		1	2	1		2	1				**9**
IteratorRMIT’Adapt.		2		1	1	1		2					**7**
Library		1			1			1	1				**4**
LocalIterator													**0**
Login		3	2		3			3	2				**13**
LoginMenu		1	1		1			1	1				**5**
LogMechanism													**0**
MedicalSpeciality		2	1						2				**5**
MedicalSpecialityR.		2	1		1			1					**5**
ObjectAlreadyInsert.		1			1			1					**3**
ObjectNotFoundExc.		1			1			1					**3**
ObjectNotValidExc.		1			1			1					**3**
Observer		4			1			1					**6**
PersistenceMech.		3			1	1		2	2				**9**
PersistenceMech’Exc.		2			1			1					**4**
PersistenceSoftExc.		1			1			1					**3**
RDBRepositoryFact.		1											**1**
RepositoryException		1			1			1					**3**
RepositoryFactory		1											**1**
RMIFacadeAdapter		3	1		1			1	1				**7**
RMIFacadeFactory		1	1	1									**3**
RMIServletAdapter		2	1		1			1	1				**6**
Schedule		1			1			1	1				**4**

continue in the next page…

Components	BS	Oct	GC	HD	TR	Tsu	KS	NN	CC	DS	DC	BC	Sum
SearchComplaintData		2	2		2			2	2				**10**
SearchDiseaseData		2	2		2			2	2				**10**
SearchHealth’Special.		2	2		2			2	2				**10**
SearchSpecial’Health.		2	2		2			2	2				**10**
ServletRequestAdapt.													**0**
ServletResponseAd.													**0**
ServletWebServer		1	1		1			1					**4**
Situation		1	1						1				**3**
SituationFacadeExc.		1			1			1					**3**
SpecialComplaint			1										**1**
SpecialComplaintSt.		1	1						1				**3**
SpecialC’StateClosed			1										**1**
SpecialC’StateOpen			1										**1**
SpecialityRep’Array		2			1			1					**4**
SpecialityRep’RDB		2			1	1		2	2				**8**
SQLPersist’Exc.													**0**
Subject													**0**
Symptom		2	1						2				**5**
SymptomRecord		3	1		1			1	1				**7**
SymptomRep’Array		2			1			1					**4**
SymptomRep’RDB		1			1			1	1				**4**
ThreadLogging													**0**
TransactionException		3			1	1	1	2					**8**
UpdateCompl’Data		2	1		2			1	1				**7**
UpdateComplaintList		1	2		1			1	1				**6**
UpdateCompl’Search		3	2		3			3	2				**13**
UpdateEmployeeData		1	1		1			1	1				**5**
UpdateEmpl’Search		1	1		1			1	1				**5**
UpdateEntryExc.		1			1			1					**3**
UpdateHealthUnit		1	1		1			1	1				**5**
UpdateHealthUnitList		3	2		1			1	2				**9**
UpdateH’UnitSearch		3	2		1			1	1				**8**
UpdateM’Speciality		1	1		1			1	1				**5**
UpdateM’List		2	2		1			1	2				**8**
UpdateM’Search		3	2		1			1	1				**8**
UpdateSymptomData		1	1		1			1	1				**5**
UpdateSymptomList		1	2		1			1	1				**6**
UpdateSymp’Search		3	2		1			1	1				**8**
Average	**0**	**1.5**	**0.7**	**0**	**0.9**	**0.1**	**0**	**1**	**0.7**	**0**	**0**	**0**	**5**

Table E-6: Changes in the components of Health Watcher

Components	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	# Changes
AbstractFacadeFactory								a			**0**
AbstractRepositoryFactory						a			1		**1**
Address	a										**0**
AddressRepositoryRDB	a									1	**1**
AnimalComplaint	a		1								**1**
AnimalComplaintState		a	a								**0**
AnimalComplaintStateClosed			a								**0**
AnimalComplaintStateOpen			a								**0**
ArrayRepositoryFactory						a			1		**1**
Command							1			1	**2**
CommandRequest							a				**0**
CommandResponse							a				**0**
CommunicationException	a										**0**
Complaint	a		1	1		1					**3**
ComplaintRecord	a										**0**
ComplaintRepositoryArray	a										**0**
ComplaintRepositoryRDB	a						1			1	**2**
ComplaintState			a	1							**1**
ComplaintStateClosed			a	1							**1**
ComplaintStateOpen			a	1							**1**
ConcreteIterator	a										**0**
ConcurrencyManager	a										**0**
ConfigRMI	a	1					1			1	**3**
ConnectionPers'Exception										a	**0**
Constants	a									1	**1**
Date	a										**0**
DiseaseRecord	a								1		**1**
DiseaseType	a										**0**
DiseaseTypeRepositoryArray	a										**0**
DiseaseTypeRepositoryRDB	a								1	1	**2**
Employee	a			1					1		**2**
EmployeeRecord	a										**0**
EmployeeRepositoryArray	a										**0**
EmployeeRepositoryRDB	a									1	**1**
ExceptionMessages	a										**0**
FacadeFactory	a							a			**0**
FacadeUnavailableException										a	**0**
FoodComplaint			1								**1**
FoodComplaintState			a								**0**
FoodComplaintStateClosed			a								**0**
FoodComplaintStateOpen			a								**0**
Functions	a										**0**
GetDataFor'DiseaseType	a	1			1		1			1	**4**
GetDataFor'HealthUnit	a	1			1		1			1	**4**
GetDataFor'Speciality	a	1			1		1			1	**4**
HealthUnit	a			1							**1**
HealthUnitRecord	a								1		**1**
HealthUnitRepositoryArray	a										**0**
HealthUnitRepositoryRDB	a								1	1	**2**
HealthWatcherFacade	a			1	1	1	1		1	1	**6**

continue in the next page…

Components	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	# Changes
HealthWatcherFacadeInit	a				r						
HTMLCode	a										**0**
HWServer								a		1	**1**
HWServlet	a	1			1		1	1	1	1	**6**
IAddressRepository	a										**0**
IComplaintRepository	a										**0**
IDiseaseRepository	a										**0**
IEmployeeRepository	a										**0**
IFacade	a			1	1				1		**3**
IFacadeRMITargetAdapter					a				1		**1**
IHealthUnitRepository	a										**0**
IIteratorRMITargetAdapter	a										**0**
InsertAnimalComplaint	a	1			1		1			1	**4**
InsertDiseaseType									a	1	**1**
InsertEmployee	a	1					1		1	1	**4**
InsertEntryException	a										**0**
InsertFoodComplaint	a	1			1		1			1	**4**
InsertHealthUnit									a	1	**1**
InsertMedicalSpeciality									a	1	**1**
InsertSpecialComplaint	a	1			1		1			1	**4**
InsertSymptom									a	1	**1**
InvalidDateException	a										**0**
InvalidSessionException	a										**0**
IPersistenceMechanism	a									1	**1**
ISpecialityRepository	a										**0**
ISymptomRepository	a								1		**1**
IteratorDsk	a										**0**
IteratorRMISourceAdapter	a										**0**
IteratorRMITargetAdapter	a										**0**
Library	a										**0**
LocalIterator	a										**0**
Login	a	1		1			1			1	**4**
LoginMenu		a					1			1	**2**
LogMechanism										a	**0**
MedicalSpeciality	a								1		**1**
MedicalSpecialityRecord	a								1		**1**
ObjectAlreadyInsertedExc.	a										**0**
ObjectNotFoundException	a										**0**
ObjectNotValidException	a										**0**
Observer				a							**0**
PersistenceMechanism	a									1	**1**
PersistenceMechanismExc.	a									1	**1**
PersistenceSoftException	a									r	**0**
RDBRepositoryFactory						a			1		**1**
RepositoryException	a										**0**
RepositoryFactory						a					**0**
RMIFacadeAdapter					a			1	1	1	**3**
RMIFacadeFactory					a			a			**0**
RMIServletAdapter									1		**1**
Schedule	a										**0**

continue in the next page…

Components	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	# Changes
SearchComplaintData	a	1			1		1			1	**4**
SearchDiseaseData	a	1			1		1			1	**4**
SearchHealthUnitsBySpecialty	a	1			1		1			1	**4**
SearchSpecialties'HealthUnit	a	1			1		1			1	**4**
ServletRequestAdapter							a				**0**
ServletResponseAdapter							a				**0**
ServletWebServer	a	1									**1**
Situation	a										**0**
SituationFacadeException	a									r	**0**
SpecialComplaint	a		1								**1**
SpecialComplaintState			a								**0**
SpecialComplaintStateClosed			a								**0**
SpecialComplaintStateOpen			a								**0**
SpecialityRepositoryArray	a										**0**
SpecialityRepositoryRDB	a								1	1	**2**
SQLPersistenceMech'Exc.										a	**0**
Subject				a							**0**
Symptom	a								1		**1**
SymptomRecord									a		**0**
SymptomRepositoryArray	a								1		**1**
SymptomRepositoryRDB									a	1	**1**
ThreadLogging										a	**0**
TransactionException	a										**0**
UpdateComplaintData	a	1	1	1			1			1	**5**
UpdateComplaintList		1					1			1	**3**
UpdateComplaintSearch	a	1		1			1			1	**4**
UpdateEmployeeData	a	1		1			1			1	**4**
UpdateEmployeeSearch	a	1					1			1	**3**
UpdateEntryException	a										**0**
UpdateHealthUnitData	a	1		1			1			1	**4**
UpdateHealthUnitList		1					1			1	**3**
UpdateHealthUnitSearch	a	1		1	1		1		1	1	**6**
UpdateMedicalSpecialityData									a	1	**1**
UpdateM'SpecialityList									a	1	**1**
UpdateM'SpecialitySearch									a	1	**1**
UpdateSymptomData									a	1	**1**
UpdateSymptomList									a	1	**1**
UpdateSymptomSearch									a	1	**1**
Total	**0**	**23**	**5**	**14**	**14**	**2**	**26**	**2**	**22**	**48**	**156**

(a) Component added.

Appendix F: Refactoring of Flat Crosscutting Patterns

The aspect-oriented refactorings presented in this appendix extend existing catalogues of refactorings with the goal of modularising crosscutting patterns. The format of the proposed refactorings follows the well-known catalogue of Martin Fowler [63]. Basically, each refactoring is presented in terms of a motivation, an abstract representation, and the mechanism, i.e., refactoring steps that should be followed to modularise the target concern. Moreover, the mechanics of our refactorings are presented in terms of fine-grained transformation steps proposed in the literature [63, 78, 83, 85, 93, 117]. Table F-1 presents the motivation (2nd column) and mechanisms (3rd column) for these fine-grained refactorings. While the complete list of refactorings for crosscutting patterns are reported elsewhere [135, 136], the rest of this section presents refactoring strategies for Black Sheep, Octopus, and God Concern.

Table F-1: Subset of fine-grained aspect-oriented refactoring steps [69, 78, 83, 117].

	Motivation	Mechanism
Move Field from Class to Intertype [117]	A field relates to a concern other than the primary concern of its enclosing class	Move the field from the class to the aspect as an inter-type declaration
Move Method from Class to Intertype [117]	A method belongs to a concern other than the primary concern of its owner class	Move the method into the aspect encapsulating the secondary concern as an inter-type declaration
Replace Extends / Implements with Declare Parents [117]	Interface implementation (or class extension) is used only when the related concern is present in the system	Replace the implements (or extends) in the class by a declare parents in the aspect
Extract Pointcut Definition [69, 83]	Part of code is related to a concern being moved to an aspect	Create a pointcut capturing the required joinpoint and context
Extract Fragment into Advice [78, 83, 117]	Part of a method is related to a concern whose code is being moved to an aspect	Move the code fragment to an appropriate advice based on a new pointcut

Mechanics of the Black Sheep Refactoring

1. *Identify black sheep in the herd*. This refactoring starts with the identification of classes with the Black Sheep concern. Concern metrics and detection strategies (Section 5.2) support this refactoring step.

2. *Steps to separate Black Sheep*. The following six steps, labelled '*a'* to 'f', aim to separate sheep slices into aspects. Some of these steps might not be applied

to specific instances of Black Sheep. They also assume a pre-existing aspect (empty or not) assigned to this concern. In case this aspect does not exist, a preliminary step would be its creation.

a. If a class implements one or more interfaces related to the Black Sheep concern, move each interface implementation to an aspect. This can be done by using the refactoring *Replace Implements with Declare Parents* [117] (Table F-1).

b. If a class extends any class only for the Black Sheep concern implementation, move the class extension to an aspect (e.g., using *Replace Extends with Declare Parents* [117]).

c. If a class has methods and attributes exclusively assigned to the Black Sheep concern, move them to an aspect (e.g., using *Move Field from Class to Intertype* [117] and *Move Method from Class to Intertype* [117]).

d. If a class has any constructor related to the Black Sheep concern, either this constructor has to be modified and used independently of the concern or it may be removed.

e. Create pointcuts to pick up necessary join points related to the Black Sheep concern during the program execution (e.g., using *Extract Pointcut Definition* [69]). If necessary, you may use *Extract Method* [63] in order to expose join points that the aspects would affect.

f. Create advice(s) to simulate the necessary concern behaviour during the program execution (e.g., using *Extract Fragment into Advice* [117]).

Mechanics of the Octopus Refactoring

1. *Identify the Octopus members*. The refactoring starts with the identification of classes which compose the Octopus parts (*body* and *tentacles*). A concern analysis implemented by our tool (Section 5.2) is used in this step to determine which attributes, methods, interface implementations, and class extensions realise the target concern.

2. *Steps to separate the Octopus body*. The following steps target at separating the Octopus body structure. There are two possibilities of classes composing the body: (a) those entirely dedicated to the Octopus concern, and (b) those mainly dedicated to this concern, but they mix it with other concerns.

    a. If one class in the Octopus body is 100 % dedicated to the concern realisation, that class can be optionally moved to a new aspect. In other words, the aspectisation is optional for classes in the Octopus body that do not realise more than one concern of interest. This decision depends on developers' judgement, but our proposed tool (Section 5.6) may support them in this choice.

    b. If one class in the Octopus body is not totally dedicated to the concern realisation, we have to separate the Octopus concern in its own component. After that, this class can be optionally moved into an aspect (see previous step).

3. *Steps to modularise Octopus tentacles*. Once the Octopus body is encapsulated into aspects, we have to restructure the Octopus tentacles. For each tentacle, it is required to apply one or more of the following steps. If no aspect exists yet, these steps may create a new aspect when appropriate.

    a. If the tentacle implements one or more interfaces related to the Octopus concern, move each interface implementation to an aspect (e.g., using *Replace Implements with Declare Parents* [117]).

    b. If the tentacle extends any class only for the concern implementation, move the class extension to an aspect (e.g., using *Replace Extends with Declare Parents* [117]).

    c. If there are methods and attributes completely assigned to the target concern, move them to an aspect (e.g., using *Move Field from Class to Intertype* [117] and *Move Method from Class to Intertype* [117]).

    d. If a class has any constructor related to the Octopus concern, this constructor has to be modified and used independently of this concern. Optionally, the constructor may be removed if it becomes unnecessary by previous aspectisation steps.

   e. If concern code is mixed inside a method, create pointcuts to pick up necessary join points related to the concern during the program execution (e.g., using *Extract Pointcut Definition* [69]). If necessary, use *Extract Method* [63] in order to expose joinpoints.

   f. For mixed concern code inside methods, create advice member(s) to introduce necessary behaviour related to the concern during the program execution (e.g., using *Extract Fragment into Advice* [117]).

Mechanics

1. *Identify the candidate concern decomposition*. Firstly, it is required to verify how new concerns with well-defined intentions can be decomposed from the God Concern.
2. *Identify the God Concern parts*. Components which centralise the main code are forming the *backbone* of this crosscutting pattern. Additionally, the other supporting components are called *arms*.
3. *Steps to modularise God Concern*.
   a. If God Concern were decomposed into new concerns, then create a new aspect for each concern when appropriate.
   b. If a component is totally dedicated to the concern, then optionally move it to an aspect. This step is not required because this component can be considered well-modularised.
   c. If a component also realises other concerns, then separate the God Concern parts and move them into an aspect.
   d. For inheritance and/or interface implementation, the refactoring *Replace Extends / Implements with Declare Parents* [117] (Table F-1) can be applied.
   e. For the concern-related attributes, *Move Field from Class to Intertype* [117] can be applied.
   f. For the concern-related methods, *Move Method from Class to Intertype* [117] can be applied.

g. For other concern-related code fragments, a combination of the two refactorings *Extract Pointcut Definition* [69] and *Extract Fragment into Advice* [117] can be applied.

Appendix G: Installation of ConcernMorph

ConcernMorph is implemented as a plug-in for Eclipse 3.3.2 using Java 1.5. It also extends ConcernMapper 2.0.3. The tool has only been used in this specific environment. Hence, there is no guarantee that it would work with different versions of Eclipse, Java, or ConcernMapper. To install ConcernMorph, the user needs to download the two compacted Zip files listed below and copy them into the plugins folder inside the Eclipse installation folder. These files are available at the ConcernMorph website (http://www.lancs.ac.uk/postgrad/figueire/concern/plugin/).

1. ConcernMapper 2.0.3, file: ConcernMapper.zip
2. ConcernMorph 1.0.0, file: ConcernMorph.zip

Basically, there are two steps required to install ConcernMorph. The first step is to unzip both files listed above. This step creates a folder for ConcernMapper and a JAR file for ConcernMorph. In the second step, the user must copy both the ConcernMapper folder and the ConcernMorph JAR file inside the ‘plugins’ folder of Eclipse. After these two steps have been executed, just open the Eclipse Platform and both ConcernMapper and ConcernMorph are supposed to be installed. Note that ConcernMorph depends on ConcernMapper in order to work properly.