THESE DE DOCTORAT
DE L'UNIVERSITE PARIS I – PANTHEON - SORBONNE

Spécialité: Informatique

Muhammad Usman Bhatti

Pour l'obtention du titre de :

DOCTEUR DE L' UNIVERSITE PARIS I -- PANTHEON - SORBONNE

Object Identification
and
Aspect Mining
in
Procedural Object-Oriented Code

Jury de Thèse:

Mme Colette ROLLAND	Directeur de thèse
M. Stéphane DUCASSE	Co-directeur de thèse
M. Kim MENS	Rapporteur
M. Yann-Gaël GUEHENEUC	Rapporteur
Mme Marianne HUCHARD	Membre du jury

The address of the author:

usman.bhatti@gmail.com

Acknowledgements

First of all, I want to thank Prof. Colette Rolland for providing me the opportunity to work under her supervision in the research group at Centre de Recherche en Informatique (CRI). She provided all the needed resources and support to complete this work. She always provided me complete autonomy to carry out my research work and always happily commented on it. I would very warmly like to thank Prof. Stéphane Ducasse. He helped me finding my way through the dense forest of thesis problematic. Despite hectic research schedule, he could always find time for discussion and overview my work. Moreover, he enormously helped me in progressing in my research, write papers and finally present this work in its current form. Had it not been for his positive attitude, this thesis might not have seen the day.

I would like to thank Prof. Kim Mens and Prof. Yann-Gaël Guéhéneuc for participating in my PhD defence committee and providing important feedback on the first version of the thesis. Yann-Gaël also provided very useful and pertinent comments on the first drafts of this thesis.

I would also like to thank Prof. Awais Rashid at University of Lancaster for useful discussions on aspect mining and crosscutting concerns. His useful advice for one phrase definition of thesis problematic aided me to clarify my research domain. Probably, this one phrase definition of thesis problematic is key for every new PhD student to define the niche of his/her work. Prof. Marianne Huchard extended her support for understanding the intricacies of Formal Concept Analysis and helping refine the Object Identification Model. She taught me to ask pertinent questions first and finding their answers later.

I worked at Diagnostica Stago for the industrial research scholarship (CIFRE). I encountered many people and learnt a lot from everyone of them. I would particularly thank Sébastien Ailleret for liberating me from industrial project constraints. This helped me enormously to concentrate on my thesis work.

All my friends at CRI, especially Assia, Olfa, Hamid, Ramzi, Hicham, Rim, Ines, and Elena with whom I shared moments of stress and joy, and profound philosophical discussion regarding every perspective of thesis and life around us.

I thank all people I met at conferences. I have had nice moments with them and also useful discussions to help me to improve my work.

Most of all, I want to thank the people beyond university and academia, whose lives were affected by this work: My parents for supporting me making this possi-

ble. My friends Asad Mehmood, Taj Khan, Omer Khayam, and Irfran Hamid for providing such a memorable company during the tough PhD years. They represent my “other life” during my stay in France, especially during weekends. And last but not least my wife for supporting and encouraging me for the timely completion of this thesis. Thank you, I could not have done this without you.

Muhammad Usman Bhatti
December 2008

Abstract

In this dissertation, we present Procedural Object-Oriented Code (POC). POC is the aftermath of the software development activity that involves state of the art object-oriented languages, without employing object-oriented analysis and design. Huge classes, absence of abstractions for domain entities, and shallow inheritance hierarchies are hallmark design defects of procedural object-oriented code. POC also consists of scattered code appearing not only due to the absence of aspects, but it also manifests scattered code appearing due to the non-abstracted domain entities *i.e.,* domain entities that do not have their proper object-oriented classes. The non-abstracted domain logic hinders mining useful crosscutting concerns related to aspects in POC. Confronted with the absence of object-oriented design and the difficulty of mining aspects in POC, we studied it from two perspectives.

First, we improve aspect mining techniques by classifying various crosscutting concerns identified in POC with a two-pronged approach: Firstly, the approach identifies and groups crosscutting concerns present in a software system: aspects as well as non-abstracted domain logic. Crosscutting concerns pertaining to non-abstracted domain entities are identified and extracted through their usage of application domain entity data. Secondly, a new metric called spread-out is introduced to quantify the divulgence of diverse crosscutting concerns.

Second, we studied the problem of object identification in procedural object-oriented code. We present a semi-automatic, tool-assisted approach for restructuring POC into an improved object-oriented design. The approach identifies principal classes in POC. These principal classes are then used to extract object-oriented abstractions using Formal Concept Analysis lattices. This is achieved by providing three different concept lattices, namely fundamental, association, and interactions views.

We developed tools to validate the approaches presented in the thesis. The approaches are validated on a recently developed industrial application. The application is used to run blood plasma analysis automatons. The results of our approach are promising.

Résumé

Dans cette thèse, nous introduisons le code orienté objet procédural (Procedural Object-Oriented Code (POC)). Ce logiciel est développé en utilisant les langages orientés objets néanmoins sans utiliser correctement l'analyse et la conception orientée objet. Grosses classes, absence des abstractions pour les entités de domaine, des hiérarchies de classes peu développées ou complètement absentes représentent les défauts clefs du POC. Le code concernant les entités absentes de domaine, appelé la "logique dispersée" de domaine, est éparpillé et embrouillé avec le code des autres classes de système qui entrave l'identification des préoccupations transverses liées aux aspects. Confronté au problème du POC, résultat de l'absence de la conception orientée objets dans le logiciel et au problème de l'identification des aspects dans ce type de code, nous avons étudié le POC de deux perspectives principales.

D'abord, pour la classification des préoccupations transverses identifiées dans le POC, nous adoptons une approche à deux axes : Premièrement, l'approche identifie et classifie les préoccupations transverses d'un logiciel : les aspects ainsi que la logique dispersée de domaine. Les préoccupations transverses concernant les entités dispersées de domaine sont identifiées et extraites par leur utilisation des entités de domaine. Deuxièmement, une nouvelle métrique appelée "spread-out" est présentée pour mesurer la dispersion des préoccupations transverses diverses.

Deuxièmement, nous avons étudié le problème d'identification d'objet dans le POC. Nous présentons une approche semi-automatique, assisté par l'outil pour la restructuration du POC. L'approche identifie les classes principales dans le POC pour le but d'unifier les données et les opérations correspondantes dans une seule classe. Ces classes principales sont alors employées pour extraire des abstractions orientées objets utilisant les treillis d'analyse de concept. Ceci est réalisé en fournissant trois différentes vues basées sur des treillis de concept, à savoir la vue fondamentale, la vue association, et la vue d'interactions de méthodes.

Nous avons développé les outils pour valider les approches présentées dans la thèse. Les approches sont validées sur le logiciel pour les automates utilisés pour faire les tests sanguins. Les résultats des approches proposées sont satisfaisant.

Contents

Acknowledgements **iii**

Abstract **v**

Résumé **vii**

1 Introduction **1**
1.1 Thesis Context 2
1.2 Problem Statement 3
1.2.1 Procedural Object-oriented Code 3
1.2.2 Restructuring Classes in Procedural Object-oriented Code 5
1.2.3 Aspect Mining in Procedural Object-Oriented Code 6
1.3 Contributions 8
1.3.1 Code Smells and Detection Strategy 9
1.3.2 Restructuring Classes in Procedural Object-oriented Code 9
1.3.3 Classification Approach and Metrics for Scattered Concerns 11
1.4 Industrial Context 11
1.4.1 Case Study: Blood Plasma Analysis Machines 12
1.4.2 Case Study Quality Metrics 13
1.5 Thesis Structure 13

2 Related Work **15**
2.1 Defects in Software 15
2.1.1 Code Smells 15
2.1.2 Object-Oriented Reengineering Patterns 16
2.1.3 AntiPatterns 16
2.1.4 Design Heuristics 17
2.1.5 Limitations of Software Defect Descriptions 17
2.2 Design Defect Detection in Software 18
2.2.1 Heuristic-based Design Defect Detection 18
2.2.2 Query-based Design Defect Detection 19
2.2.3 Design Defects Taxonomy 20
2.2.4 Visualizations for Design Defect Detection 20
2.2.5 Other Tools 21

2.2.6 Limitations of Design Defects Detection 22
2.3 Object Identification and Class Restructuring 22
2.3.1 Object Identification in Procedural Code 23
2.3.2 Object-Oriented Restructuring 26
2.3.3 Limitations of object-oriented Restructuring 28
2.4 Identification of Crosscutting Concerns 28
2.4.1 Crosscutting Concerns 28
2.4.2 Aspect Mining 30
2.4.3 Concern Quantification 35
2.4.4 Aspect Refactoring 36
2.4.5 Limitations in Crosscutting Concerns Identification 36
2.5 Discussion 36
2.5.1 Design Defects and Code Smells 37
2.5.2 Object-Oriented Restructuring 37
2.5.3 Aspect Mining 38
2.5.4 Proposed Solution 38

3 Procedural Object-Oriented Code **41**
3.1 Overview 41
3.2 Background — Object-Oriented Paradigm 42
3.3 Procedural Object-Oriented Code 43
3.4 POC Design Defects and Code Smells 45
3.4.1 Missing Domain Entities 47
3.4.2 Shallow Inheritance Hierarchies 48
3.4.3 Missing Types 51
3.5 Detection of POC Design Defects and Code Smells 54
3.5.1 Detecting Scattered Code in POC 56
3.5.2 Proposed Approach — Scattering Analyzer 57
3.5.3 Discussion 58
3.6 Conclusion 59

4 Reconsidering Classes in POC **61**
4.1 Overview 61
4.2 Formal Concept Analysis 62
4.3 Motivation 62
4.3.1 Goals of our Intended Model 63
4.3.2 Current FCA-based Techniques 63
4.4 Object Identification in POC 64
4.4.1 Identification of Principal Classes 64
4.4.2 Principal Class Compositions 66
4.4.3 Hierarchical Method-Attribute Relationship 67
4.4.4 The case of Enumerated Types 73
4.5 Discussion 75
4.6 Conclusion 76

5 Scattered Concerns in POC **79**
5.1 Overview . . . 79
5.2 Aspect Mining in Procedural Object-Oriented Code . . . 80
5.2.1 Aspect Browser . . . 80
5.2.2 Aspect Browser Results . . . 82
5.2.3 FAN-in Metric . . . 83
5.2.4 FAN-in Results . . . 84
5.2.5 Comparison of Results . . . 86
5.2.6 Taxonomy of Crosscuttingness in POC . . . 87
5.3 Discussion . . . 90
5.4 Conclusion . . . 91

6 Concern Classification in POC **93**
6.1 Overview . . . 93
6.2 Concern Classification . . . 94
6.2.1 Model for Concern Classification . . . 96
6.2.2 Domain Entity Concern Assignment . . . 98
6.2.3 Algorithm for Concern Classification . . . 99
6.3 Scattering Metrics of Crosscutting Concerns . . . 99
6.4 Discussion . . . 101
6.5 Conclusion . . . 102

7 Tools and Validation **105**
7.1 Scattering Analyzer . . . 105
7.1.1 Identifier Analysis . . . 106
7.1.2 Identifier Results . . . 106
7.1.3 Fan-in Metric . . . 107
7.1.4 Fan-in Results . . . 107
7.1.5 Discussion . . . 108
7.2 Reconsidering Classes: Application of the Approach . . . 108
7.2.1 Tool Support . . . 109
7.2.2 Validation of the Approach . . . 111
7.2.3 Discussion . . . 115
7.3 Classifying Crosscutting Concerns . . . 116
7.3.1 Validating Concern Classification Approach . . . 116
7.3.2 Scattering Metrics of Crosscutting Concerns . . . 117
7.3.3 Discussion . . . 120
7.4 Conclusion . . . 121

8 Conclusion and Perspectives **123**
8.1 Contributions . . . 123
8.2 Future Work . . . 125

A Sommaire **129**
A.1 La Problématique 130
A.1.1 Code orienté objet procédural 131
A.1.2 La restructuration des classes dans le COP 133
A.1.3 Identification d'aspect dans le COP 134
A.2 Contributions 136
A.2.1 Mauvaise odeurs et leur détection 137
A.2.2 L'approche pour la restructuration des classes 137
A.2.3 L'approche pour la classification des préoccupations transverses 139

B Introduction to Formal Concept Analysis **141**
B.1 Introduction 141
B.2 Context and Concepts 142
B.3 Concept Lattice 143

Chapter 1

Introduction

Another flaw in the human character is that everybody wants to build and nobody wants to do maintenance.

Kurt Vonnegut, Hocus Pocus.

Companies always try to look for means to reduce software development cost because this cost does have a direct effect on their competitiveness. The reuse of software components amplifies software developer's capabilities because the developers do not need to develop the components anew. The reuse, thus, reduces the overall cost of software development. Software reusability builds on a good software structure through a comprehensive application of software design heuristics, models, and guidelines [GHJV95, Mey88, Rie96]. These models and guidelines advocate software modularization, *i.e.,* the division of large components into smaller, autonomous, and manageable modules. Each of the modules should address a smaller part of the application domain, hence each of the modules can be evolved and maintained independently. Software modularity is strongly dependent upon the principle of information hiding and data encapsulation, which state that all the information about a module should be private to the module unless specifically declared in its public interfaces [Par72]. Respecting these principles ensures that knowledge pertaining to a particular module is encapsulated inside the module boundaries. Thus, a small change in software system specifications triggers changes in just one module, or a small number of modules, because the changes are localized to the private portions of the module. The changes in the module do not impact its public interfaces and consequently other dependent modules.

Object-oriented languages provide support for well-modularized software where all the knowledge of domain concepts is encapsulated in their corresponding classes. These classes contain the state and a set of operations related to a particular domain concept. Hence, the code related to a domain concept, or a domain entity, is encapsulated in its particular class in software. The functionality of these classes is exposed to their client classes through well-defined interfaces. Thus, clients are oblivious of the implementation details of a class. Changes to a class are confined

to the code residing inside the class. Therefore, a good object-oriented design leads to software reusability and reduces software costs.

Recent works on software modularity consider software as a collection of software concerns [Kic96, TOHJ99]. A concern is defined as “any matter of interest in a software system” [FECA05]. It has been reported that by their very nature, some software concerns are difficult to modularize using object-oriented or predecessor software development paradigms and are termed as *Crosscutting Concerns* [Kic96]. Crosscutting concerns are scattered and tangled across application classes. Hence, crosscutting concerns violate information hiding principle and this violation of information hiding severely affects software modularity because changes to these crosscutting concerns are not bounded to one class but they are scattered to various class in an object-oriented software system.

Aspect-Oriented Programming (AOP) proposes language constructs to encapsulate crosscutting concerns [KLM+97]. Aspect mining tools and techniques have been proposed [KMT07] to facilitate the task of crosscutting concerns identification in non-AOP code. The identified crosscutting concerns are then refactored in aspects to improve software modularity. For correct aspect identification, it is important that the crosscutting concerns identified in non-AOP code are correctly associated to the absence of aspects. The correct aspect identification is essential to correctly identify candidates aspects that can be refactored into AOP constructs [HRB+06].

1.1 Thesis Context

We hypothesize in this dissertation that *Procedural Object-oriented Code* (POC) appears when object-oriented software does not exhibit good object-oriented design. Procedural object-oriented code manifests itself in the code in the form of various design design defects and code smells. Design problems occurring in procedural object-oriented code also give rise to scattered code. This scattered code complicates the detection of crosscutting concerns in procedural object-oriented code. This problem occurs because the techniques employed for the detection of crosscutting concerns in POC also report scattered code related to object-oriented design problems in their results. Hence, it becomes difficult to relate the scattered code as related to the absence of object-oriented design or the absence of aspects.

Therefore, two main problems to be solved in procedural object-oriented code are: (1) *classification of crosscutting concerns identified in the context of POC* and (2) *restructuring of classes to rectify design problems*.

For the classification of the scattered code related to crosscutting concerns in POC, we demonstrate that the scattered code identified through aspect mining techniques can be classified as pertaining to the absence of objects or aspects based on the usage of domain entity data and scattering metrics. For the restructuring of classes in POC, we describe an approach through the usage of Formal Concept Analysis.

1.2 Problem Statement

In this problem statement, we state that software systems developed using object-oriented languages sometimes show an absence of a proper design. These software systems cannot be reused, changed or maintained without incurring high costs. The lack of object-oriented design occurs because object-oriented analysis and design process is only partially applied to speedup the software development process. In addition, extensive maintenance activities dilute software design over a period of time. The lack of design causes the violation of the principles of encapsulation and information hiding in object-oriented software systems.

The absence of object-oriented design in software systems result in certain design defects and their manifestation in programs in the form of code patterns. Consequently, application classes are less cohesive and more coupled because of the scattered code related to the design defects. Thus, a change in specifications creates modification ripples in several classes [Mey88]. Moreover, the scattered code appearing due to the absence of object-oriented design also complicates the problem of crosscutting concerns identification.

Nevertheless, software cannot be discarded due to these design problems because, the software, in itself contains valuable artifacts that encompass domain knowledge and sometimes it is all too expensive to develop a new software from scratch. Software Reengineering and Restructuring aim to improve or transform existing software so that it can be understood, controlled and used anew [DDN02]. By restructuring software, the design defects in programs can be removed and software systems can be evolved for future needs easily [Cas98, RW98]. Software restructuring extends the lifetime of software, thus increasing the return of investment of their owners.

In the following sections, we describe various research questions that arise from the absence of object-oriented design in software. We summarize pertinent existing research works to demonstrate their limitations vis-à-vis the proposed questions. Later, we describe our contributions that propose answers to these research questions.

1.2.1 Procedural Object-oriented Code

> *What are the main characteristics of the classes and the code patterns that appear due to the absence of object-oriented design?*

We call the software systems developed using the state of the art object-oriented languages, nevertheless demonstrating the absence of object-oriented design as *Procedural Object-oriented Code* (POC). We believe that it is important to identify the design defects and code smells appearing in POC so that these can be identified and detected, and later removed from the code.

Procedural object-oriented code consists of *Partially Decomposed* classes whereby huge classes define logic for subsystems or services instead of particular

domain entities. The partially decomposed classes in POC appear because of the incomplete application of object-oriented decomposition or extensive maintenance activities. These partially decomposed classes in POC result in certain architectural-level design defects. These design defects include the absence of class hierarchies. Consequently the type and subtype relationships are missing for domain entities *i.e.,* domain entity code is not produced in a hierarchical relationship represented by a parent class and its derived subclasses. Thus, the code related to a domain entity cannot be reused through specialization, a leverage obtained with the presence of hierarchical relationships. In addition, certain domain entities are not represented in their precise classes but their code is scattered across the other classes making up a system.

Figure A.1, which shows an *inheritance* hierarchy graph, demonstrates that in POC classes are often huge structures with limited use of inheritance. The huge classes and a scarcity of inheritance links between classes points towards the missing well-decomposed classes and class hierarchies for the domain entities supported by the system.

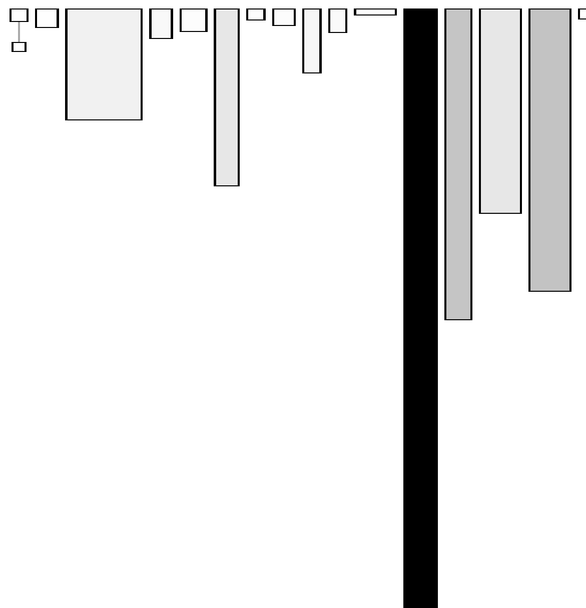

Figure 1.1: *Procedural Object-Oriented Code* — Rectangles represent classes, edges represent the inheritance relationships between classes, height and width of rectangles are dependent upon number of methods and number of attributes respectively, and node color is determined by the number of lines of code (a snapshot taken using the MOOSE reverse engineering environment [MGL06]).

The higher-level design defects in POC manifest themselves in code as various code patterns, or commonly called code smells. These code smells include the occurrence of cloned calls in methods, cloned template code, global enumerated

types, and misplaced methods. These code smells should be presented in detail so that their manifestation in code be identified, detected, and corrected for an improved object-oriented design.

Design defects are problems occurring from bad design practices, or deviations from well-known design norms [DM00, Rie96]. There are some pertinent studies that mention the code and architectural patterns that describe design problems in code [BMMM98, DDN02, FBB+99, Rie96]. However, the architectural patterns that occur because of the missing object-oriented design only describe their high-level, architectural manifestation [BMMM98]. No code-level examples are provided. Code smells describe small design anomalies in code that represent refactoring opportunities [FBB+99]. But the described code smells provide a list of design anomalies that affect a few portions in code. These do not provide the code smells related to the absence of an overall object-oriented design in programs.

> *Is it possible to provide techniques and tools to ease the discovery of POC classes and the code patterns appearing in them?*

There are several works that deal with the identification of problematic patterns in code. Existing software quality metrics and visualization techniques do provide clues for the identification of partially decomposed classes and lack of inheritance hierarchies [Ciu99, LD03, Mar04, MlHG06]. Nevertheless, these do not support the identification of the POC code smells that result in scattered code occurring due to the absence of object-oriented design such as cloned method calls and global enumerated types [BD07]. Their detection requires the use of structural analysis of the scattered entities and their behavior. Therefore, the existing tools need to be improved to identify all of the POC code smells.

1.2.2 Restructuring Classes in Procedural Object-oriented Code

Procedural object-oriented code consists of partial decomposed classes that encapsulate logic for several domain entities and hence some of the domain entities do not have their object-oriented abstractions, *i.e.,* classes in the code. We believe that the design defects and code smells in POC have an adverse effect on software modularity and these design defects and their associated code smells should be removed from object-oriented programs. Thus, it is important to look for ways to restructure classes in procedural object-oriented code to encapsulate each of the domain entity into their appropriate class so that the software modularity of POC is enhanced.

> *Can we extract meaningful object-oriented classes and class hierarchies from POC classes that represent an improved object-oriented design?*

Object identification techniques have been proposed in literature to detect objects in non-OO programs to transform these programs into object-oriented programs [CCDD99, CCM96, NK95, SLMM99, SR99, vDK99a]. These techniques

employ Formal Concept Analysis (FCA) to detect objects in procedural program by grouping global variables and functions that operate upon these variables. These techniques however do not take into account the partially decomposed classes present in procedural object-oriented code and other forms of object-oriented artifacts (enumerated types and method calls) to improve object identification algorithms.

Another set of proposals regarding restructuring of object-oriented classes pertains to the use of FCA to understand object-oriented programs and refine class hierarchies [ADN05a, Moo96, ST97, SS04]. However, these techniques propose to search for and correct small class hierarchy anomalies in object-oriented programs. These approaches do not target to fix architectural design defects in object-oriented programs. Thus, these techniques are not applicable to migrate POC to an improved object-oriented design.

One more set of proposals proposes code refactoring through the *manual* identification of *small* design problems within class hierarchies and provide various heuristics for their rectification [DDN02, FBB+99]. However, restructuring of classes in procedural object-oriented code following these guidelines is too cumbersome because these techniques only propose to improve object-oriented classes without putting overall object-oriented design into question.

Moha *et al.* [MHVG08] define a very similar approach to the work presented in this thesis. The approach suggests the use of Relational Concept Analysis for removing AntiPatterns [MHVG08] in code. However, the approach when applied on POC produces huge lattices cluttered with too much information that cannot be used to extract useful information from POC.

In summary, the existing approaches for object identification in procedural programs, and class hierarchy reengineering and restructuring in object-oriented programs are not apt for class hierarchy inference from POC. These approaches when applied on POC produce huge lattices that do not allow the interpretation of useful object-oriented concepts. Moreover, these approaches do not take into account the code smells occurring in POC (cf. Section 1.2.2 and Chapter 3).

Hence, there is a need to define an approach to restructure classes in POC. This approach should integrate object-oriented features and the POC code smells to obtain an improved object-oriented design from POC classes.

1.2.3 Aspect Mining in Procedural Object-Oriented Code

Crosscutting concerns result in scattering and tangling of code with various other concerns of software system. Crosscutting concerns discovery is a difficult task and for the purpose of aspect discovery in programs, aspect mining tools and techniques have been proposed [KMT07]. Aspect Mining is a reverse engineering technique. It automates the process of aspect discovery in non-AOP software systems. These techniques rely on the assumption that any scattered code in systems is related to crosscutting concerns and should be refactored into aspects [CMM+05, HK01]. In outcome, these techniques propose their user one or more aspect *candidates* based

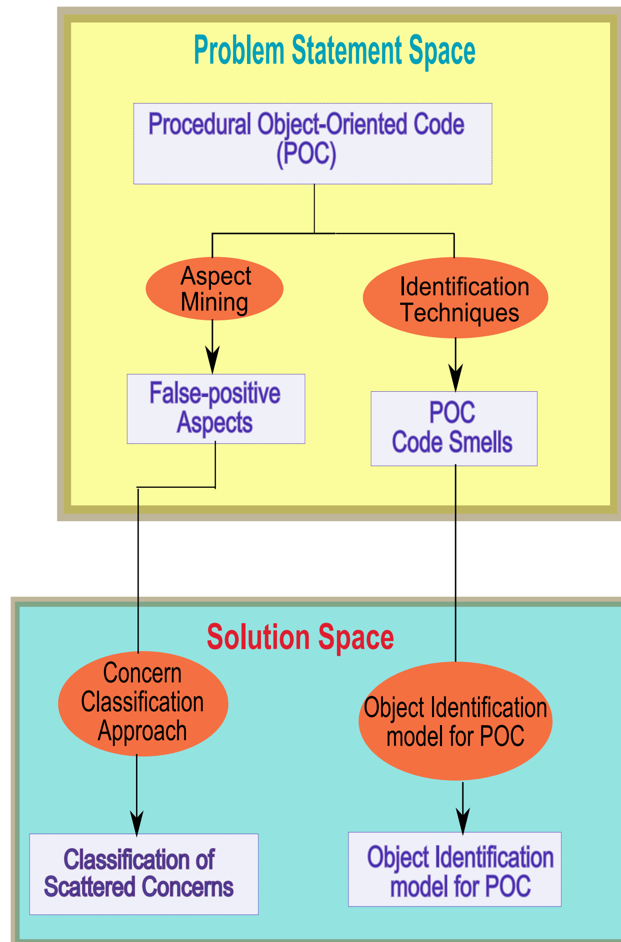

Figure 1.2: Thesis Plan

on lexical information of the code, and static or dynamic analysis.

However, in the context of procedural object-oriented code, aspect mining techniques do not provide reliable results [BD08]: The results contain non-aspect candidates, or false-positive aspects. The problem of false-positive aspects occurs because procedural object-oriented code consists of scattered and tangled code appearing not only due to the absence of aspects, but it also manifests scattered code appearing due to the absence of object-oriented design. The lack of object-oriented design gives rise to non-abstracted domain entity code *i.e.,* the scattered code related to the domain entities that do not have their own classes. Therefore, we observe that proposed aspect mining tools, when applied on procedural object-oriented code result in the identification of missing domain entities in the list of crosscutting concerns [BD08]. These domain entities are falsely identified because aspect mining techniques assume that scattering and tangling only originate

from missing aspects, while other domain-related concerns have been encapsulated into their object-oriented abstractions. As the phenomenon of good design is not present in procedural object-oriented code, therefore, we are confronted with the problem of the qualification of crosscutting concerns identified by aspect mining tools as aspects.

> *Can we differentiate scattered code appearing in POC and distinguish the scattered code occurring due to the absence of objects from the list of identified aspects for correct aspect identification?*

Current aspect mining techniques [KMT07] do not report false-positives appearing due to the non-abstracted domain logic. Other concern-based studies [EZS+08, KSG+06, RM02] provide a general approach for crosscutting concerns identification without mentioning the scattered code appearing due to the absence of design.

For correct aspect identification in POC, it is important to study the characteristics of the scattered code appearing due to the different reasons. It will help define a strategy that classifies scattered and tangled code as related to missing objects or aspects and we believe that POC provides a good opportunity to classify scattered code appearing in object-oriented programs. This classification will help the current aspect mining techniques to differentiate between the different types of scattered code. Thus, aspect mining techniques will better help developers to distinguish between scattered code appearing from the POC design defects and the absence of aspects. This distinction in scattered code permits to apply appropriate refactorings to encapsulate the scattered code.

The overall problem space for this report is illustrated in Figure A.2.

1.3 Contributions

The research questions detailed above are all pertaining to the absence of object-oriented design *i.e.,* POC. Since no prior work reports POC from code scattering and class restructuring perspectives, thus as we briefly mention in this chapter and as we shall show in the thesis, the existing literature does not answer these questions.

We have provided solutions for two independent problems in POC. The first solution pertains to the occurrences of scattered code in POC and its classification. We choose this perspective because the scattered code originating from the absence of both objects and aspects in POC provides an excellent opportunity to study the nature of code scattering. The purpose of the proposed approach is to propose an enhancement of the existing aspect mining techniques so that these tools can distinguish the scattered code related to the absence of aspects from that of the absence of objects. The second perspective is the restructuring of POC classes into an improved object-oriented design that consists of more cohesive classes. This perspective is important to look for a strategy to transform POC into a useful object-oriented design.

We do not correlate the two solutions presented in this dissertation. We believe that for the correlation of the two perspectives regarding classification of scattered code in POC and restructuring of classes should form the subject for another thesis.

In the section below, we list the main contributions of the thesis. These contributions provide an answer to the research questions formulated above. We briefly list the contributions of this thesis below before providing their detailed account.

- Description of the POC design defects and code smells along with their identification strategy. We define a tool based on the principles of scattered code identification to detect code smells that appear scattered in code.
- An approach for restructuring classes in POC towards an improved object-oriented design.
- Classification of crosscutting concerns found in the context of POC through structural analysis and metrics.

These are illustrated in Figure A.2 and we detail these in the following sections.

1.3.1 Code Smells and Detection Strategy

A list of the design defects and consequent code smells related to procedural object-oriented code has been elaborated in this thesis. We list the design defects and their associated code smells in POC. The design defects are already mentioned such as the scarcity of inheritance hierarchies and huge classes. They result in the POC code smells that include Common Calls and global enumerated types. For the detection of the design defects and code smells in POC, we identify two groups. The first group consists of those design defects and code smells that demonstrate similar symptoms as the existing set of design defects and code smells. These design defects and code smells can be detected through the use of existing software quality metrics. The code smells in the other group cannot be detected using the quality metrics because these result in scattered code. So, for their detection, we proposed the usage of the techniques for the detection of scattered code. For this purpose, we present an approach that searches for scattered code through the usage of Identifier analysis and Fan-in analysis.

The work presented in this thesis is the first research work that provides a catalogue of the design defects and the code patterns that appear in the absence of object-oriented design, and consequently, the first detection strategy to identify these characteristics and patterns. Moreover, the work presented here is the first work that reports the scattered code resulting from the absence of object-oriented design and its detection strategy with aspect mining techniques.

1.3.2 Restructuring Classes in Procedural Object-oriented Code

We present a semi-automatic, tool-assisted approach for restructuring classes in the code showing signs of absence of object-oriented design [BDH08]. The approach

is based on Formal Concept Analysis (FCA) [GW99]. Our approach helps inferring coarse-grained class hierarchies from the existing set of classes in procedural object-oriented code. The overall approach is illustrated in Figure A.3 and it is described below.

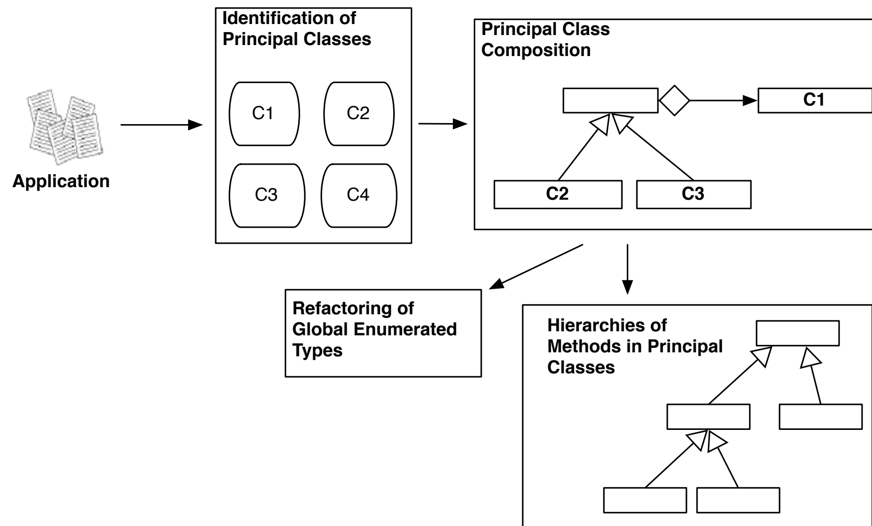

Figure 1.3: An Approach for Restructuring Classes in POC

1. The first step in our approach is the discovery of cohesive groups of methods and attributes in application, following certain rules, to decompose large classes. This decomposition of classes is achieved by looking at methods present in the code and user-defined types upon which they operate. These cohesive groups are called *principal classes*.

2. An object-oriented architectural abstraction for principal classes is obtained to obtain the interaction and composition relationships of principal classes amongst themselves. This architectural abstraction is achieved by searching for *create-create* pattern *i.e.,* classes that instantiate other classes.

3. Hierarchical abstractions for the methods and attributes of each of the principal classes are obtained by analyzing their accesses to the individual elements of user-defined types. These abstractions are obtained through various contexts of FCA lattices. We define three views: *Fundamental View*, *Association View*, and *Common-Interaction View*. A reengineer can abstract hierarchies for domain entities by inferring the hierarchical information from these views.

4. Scattered code related to global enumerated types is identified and refactored into new methods. These methods are then added to the user-specified principal class.

As discussed earlier, the existing approaches for class hierarchy reengineering and restructuring are not apt for class hierarchy inference from POC. Our approach fills the gap as it produces lattices for each of the principal classes identified by the approach. The reduction of FCA lattices is achieved because the principal classes only contain state and methods related to a particular domain entity. Hence, class information that is introduced into the lattices is reduced in a meaningful way. Therefore, lattices are smaller and class hierarchies are easier to interpret.

1.3.3 Classification Approach and Metrics for Scattered Concerns

We evaluate two aspect mining techniques on an industrial system and report a new set of false-positive aspect candidates identified by aspect mining techniques [SB]. We observe that the current aspect mining techniques are insufficient to distinguish the scattered code resulting from the absence of objects from the scattered code that appears due to the absence of aspects. This limitation of aspect mining techniques occurs because aspect mining techniques relate scattered code to aspects regardless of the fact that the scattered code appears due to the absence of certain abstractions or inherent limitations of OO mechanisms to encapsulate crosscutting concerns. We briefly describe a taxonomy of aspect mining tools regarding their abilities to detect scattered concerns related to scattered types and scattered behavior [BD08].

A classification approach is proposed for the code that appears due to the absence of objects to distinguish it from the candidate aspects [BDR08]. The classification takes a two-pronged approach. Firstly, the approach identifies and groups crosscutting concerns present in a software system: aspects as well as non-abstracted domain entities. Crosscutting concerns pertaining to non-abstracted domain entities are identified and extracted through their usage of application entity data. Secondly, a new metric called *spread-out* is introduced to quantify the scattering of diverse crosscutting concerns.

Our work is the first work in aspect mining that reports the occurrences of false-positive aspect candidates identified by aspect mining techniques due to the absence of object-oriented design [BD08]. Thus, the classification approach presented in this thesis helps aspect mining tool developers to identify and filter from the results of their tools those candidates that appear due to the absence of classes for domain entities. The filtering of false-positives from aspect mining results permits to perform aspect refactoring correctly.

1.4 Industrial Context

The work presented in this thesis was carried out during the research work at Diagnostica Stago. Diagnostica Stago manufactures and markets a range of instruments of blood analysis. It develops and provides a product line of blood analysis automatons for research as well as for simple blood analysis. The automatons are composed of two main subsystems: a hardware subsystem consisting of mechan-

ical and electronic parts like, arms, drawers, and a software subsystem managing the hardware and carrying out blood plasma analyses. The user of the automaton loads one or more tubes containing blood plasma, as well as products (reagents), in the drawers of the automaton, associates analyses to be performed on each tube, and launches the analyses. The automaton performs the analyses for blood-related diseases and results are calculated. The overall process is depicted in Figure 1.4.

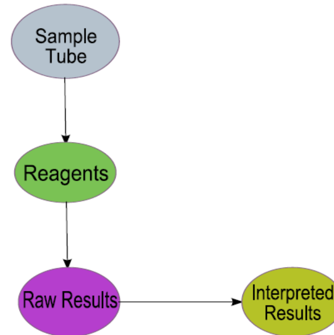

Figure 1.4: Testing Plasma Tubes

1.4.1 Case Study: Blood Plasma Analysis Machines

We selected as our case study a recently developed software system that is deployed on the latest blood plasma analysis automaton. The software is written in the C# programming language [HWG06], and hence adheres to the object-oriented paradigm. Software comprises of components that provide all the above described functionalities of analysis automatons.

For the sake of precision and clarity, we shall only present the software subsystem that manages the business objects and operates with the database layer to manage the data associated with these objects. This is one of the software subsystems that is replicated on each new machine and, in case of any design problems in the subsystem, it is important to remove them. The improvement in the design of this subsystem will mean a good amount of software reuse and costs cut in the development of software for new machines.

Certain core functionalities, such as blood analysis data, reagents used by the machines, results and patient data are the key features implemented at this subsystem. Every test is performed on patient data and the results of the tests are then stored in persistent storage system. The persistent storage system is extensively used to record all the business objects, machines activities, test traceability information, machine products, and machine maintenance information. Quality control is performed on the machines with plasma samples for which the results are known beforehand, to determine the reliability of machine components. In addition for quality control purposes, a machine is calibrated with the predetermined plasma samples so that results can be displayed in different units according to the needs

of a biologist or a doctor for easy interpretation. Blood analyses are performed on patient's blood plasma in hospitals or laboratories.

Table 1.1: Case Study Metrics

Component Name	LOC	NOM	NOA	DIT	LCOM
CPatient	11,462	260	9	1	0.85
CTest	2792	81	13	1	0.72
CProduct	2552	77	6	1	0.72
CResults	1652	52	13	1	0.85
CPersistency	1325	67	29	2	0.97
CGlossary	1010	121	5	1	0.80

1.4.2 Case Study Quality Metrics

Table 1.1 present the design quality metrics [HS96] of the subsystem described above. Lines Of Code (LOC) tallies all lines of executable code in the system. Number Of Methods (NOM) and Number Of Attributes (NOA) metrics indicate respectively the total level of operations implemented and the amount of data maintained by the class. Depth Of Inheritance (DIT) indicates the level of inheritance of a class. Finally, Lack of Cohesion Of Methods (LCOM) indicates the cohesion of classes constituents by examining the number of disjoint sets of the methods accessing similar instance variables; lower values indicates better cohesiveness.

These metrics are provided to depict an overall picture of the subsystem that we shall study during the course of this thesis. The approaches that we shall propose in the thesis are validated on the subsystem.

1.5 Thesis Structure

The report has been organized as follow:

- Chapter 2 describes the existing state of the art on design defects and code smells appearing in object-oriented programs and their proposed restructuring techniques. The chapter also provides the state of the art on aspect mining and concern quantification techniques. The chapter provides a discussion of the limitations of the state of the art vis-à-vis our problem statement.

- Chapter 3 provides a detailed insight into the design defects and code smells appearing in POC. An approach is proposed based on the identification of scattered code to detect the code smells resulting in the scattered code in POC.

- Chapter 4 describes our approach for restructuring classes in POC into an improved object-oriented design. We apply this approach on POC to extract classes through the use of Formal Concept Analysis.

- In Chapter 5, we evaluate procedural object-oriented code through the prism of aspect mining techniques. We employed two aspects mining techniques to find crosscutting concerns present in POC. We demonstrate that, in POC, there is a huge percentage of false-positive aspects in the results of aspects mining tools. A taxonomy of various crosscutting concerns is also described for POC.

- Chapter 6 provides an automated classification approach. This classification approach is used to classify scattered code in POC. We classify scattered code as related to objects and aspects. Moreover, an evaluation of scattering metrics is presented. We present a new metric called *spread-out* of crosscutting concerns to quantify scattering of various concerns.

- Chapter 7 describes the validation of the approaches proposed in this thesis. We discuss the results of the tools along with their limitations.

- Chapter 8 concludes the dissertation and presents future research directions. We identify research directions that we could not explore due to the lack of time and resources. We intend to explore these directions in our future work.

- Appendix B is a complement to the thesis and explains in detail the mathematical background of Formal Concept Analysis.

Chapter 2

Related Work

In this chapter, we present the work related to the research proposals presented in this thesis related to POC, object-oriented restructuring, and concern classification. We present software defects, namely design defects and code smells, which are deviations of object-oriented design norms and various detection techniques to detect these software defects. After presenting the software defects, we present the techniques that are proposed to restructure object-oriented programs to remove design problems appearing due to the software defects.

Later, detection techniques for crosscutting concerns are discussed to bring forth their limitations vis-à-vis detection of scattered code related to the absence of design in POC. We present an extensive literature survey of the existing techniques and their limitations vis-à-vis classification of different concerns.

2.1 Defects in Software

Software defects are common, recurrent design and implementation problems that appear in object-oriented software. Generally, they are deviations from the well-known design norms [Mar00, Rie96]. We distinguish two types of defects: Higher-level design defects and code-level problems. Higher-level design defects are commonly associated to AntiPatterns [BMMM98]. However, the term design defects remains a more general term. Code-level problems in software are generally known as Code Smells [FBB+99]. Code smells are symptoms of the presence of higher-level design defects. In the section below, we list various research works that discuss the presence of design defects and code smells.

2.1.1 Code Smells

Fowler *et al.* [FBB+99], in their pioneering work, introduced the concept of Code Smells. A code smell represents a low-level design problem. Hence, code smells depict refactoring opportunities to improve the software design for easy maintenance and reuse. The seminal book on code smells presents a popular catalogue

of 22 code smells and steps for their possible refactorings [FBB+99]. Code smells include duplicate code, long method, large class, long parameter list, etc. Possible remedies include encapsulate duplicate code in a helper method, breaking up the long method by extracting some instructions in a new method, and moving methods and fields within class hierarchies. The work on code smells assumes that the program demonstrating code smells has a good overall object-oriented design. The program only suffers from code problems that are localized and do not span the overall program code. As code smells are localized to certain program parts, they can be removed by manually applying the proposed refactorings at the problematic locations. The work on code smells does not discuss the code smells that span all the program classes appearing due to the complete absence of class hierarchies and the absence of object-oriented encapsulation for domain entities.

2.1.2 Object-Oriented Reengineering Patterns

Object-oriented reengineering patterns present the reengineering of legacy systems in a pattern format, discussing the pros and cons of various steps involved in reengineering [DDN02]. Patterns related to reverse engineering, software redesign, and problem detection and correction are discussed. For the problem description part, the authors carry forward work from the aforementioned code smells and provide restructuring strategies in pattern-oriented format. Hence, the overall process of code smells removal is presented from the reengineering perspective. Problem solving patterns for design problems like large classes, null objects, navigational code, and transform conditional to polymorphism are discussed in depth. The work on reengineering patterns does not focus on the reengineering of object-oriented systems lacking an overall object-oriented design.

2.1.3 AntiPatterns

Brown *et al.* present higher-level, architectural AntiPatterns [BMMM98]. AntiPatterns describe bad solutions to recurrent design problems that result in negative consequences for software quality. Contrary to design patterns [GHJV95], AntiPatterns describe what should not be done. AntiPatterns generally describe a design defect at a higher-level of software design without concretely presenting the associated code patterns that provide a precise code-level manifestation. This way, they remain more general in their nature, and hence, a large part of the identification of AntiPatterns lies with the skill of the reengineers. A reengineer has to correctly interpret AntiPatterns according to the context of his system to remove them from their programs.

According to the context of the work presented in this thesis, we detail three AntiPatterns related software design, namely The Blob, Functional Decomposition, and Spaghetti Code [BMMM98]. These all describe the design defects due to the absence of proper object-oriented design in software and their consequences. The Blob AntiPattern describes the presence of a large controller class that works on

the data described in the surrounding data classes. Functional Decomposition describes the code written in a process-oriented manner with the existence of a main routine that calls numerous subroutines. Spaghetti Code represents classes with no structure and the presence of long methods working on global variables. No inheritance and polymorphism is present in Spaghetti code. In summary, software design AntiPatterns do represent some form of the manifestations of the absence of object oriented design. The limitation of AntiPatterns is that they only describe high-level, architectural anomalies; they do not present any concrete code-level patterns that appear due to the presence of AntiPatterns. Detection techniques for AntiPatterns describe AntiPatterns as a set of code smells for their detection in code [MGMD08].

2.1.4 Design Heuristics

Reil [Rie96] describes 61 heuristics that characterize a good object-oriented design. These design heuristics allow developers to manually evaluate the design of their object-oriented applications. The author also mentions two design defects that are incarnated in the form of huge classes: Behavioral god class and Data god class. The first one describes a huge, god class (behavioral form) where the class "performs most of the work, leaving minor details to a collection of trivial classes". It is similar to Functional Decomposition AntiPattern. The second design defect describes a huge class that encapsulates all the data attributes that are needed by other classes for their functioning.

2.1.5 Limitations of Software Defect Descriptions

Code smells and AntiPatterns describe the software defects encountered by the authors while developing and maintaining software. But these do not enlist an exhaustive list of code problems that can occur in software systems. We are interested in finding the design defects that appear in the absence of object-oriented design and their consequent code smells.

AntiPatterns provide a list of design problems that can occur due to problems in object-oriented design. Nevertheless, their definition is abstract and a few details of their manifestation in code are described such as long methods and huge classes. However, design defects similar to AntiPatterns can have a different manifestation in code. An example that we shall elaborate later is related to the Blob AntiPattern. Although, the Blob describes scattered behavior related to Data classes, the scattered behavior in case of the Blob AntiPattern resides only in a single class. However, in the absence of an overall object-oriented design, the behavior related to Data classes may not reside in a single class but in multiple god classes. Therefore, AntiPatterns do not present all forms of design defects in a missing object-oriented design.

Code smells on the other hand describe small design anomalies that affect a few portions in code. This is because while describing these code smells the authors do

not put the overall object-oriented design into question. Hence, it is believed by the authors that manual refactoring in a few number of classes can remove these code smells from code as they do not provide tools and techniques for their detection. We shall later show that a degenerated object-oriented design may demonstrate similar design defects as suggested by some AntiPatterns. But these similar design defects give rise a new code smells that are different from the existing set of object-oriented code smells [FBB+99].

2.2 Design Defect Detection in Software

Code smells and AntiPatterns provide a textual description of software defects, which is destined for experienced developers. However, their detection in software remains a manual activity. For the purpose of their identification in programs, detection techniques have been proposed to automate the task of identification of design defects and code smells. Below, we present approaches that aim to automate the task of software defect identification in code.

2.2.1 Heuristic-based Design Defect Detection

Munro [Mun05] noticed the limitations of textual description of code smells and proposed a template to describe code smells more systematically. The template consists of three parts: code smell name, textual description of code smell features, and detection heuristics. The work contributes to a more precise specification of cod smells than their textual description. For detection purpose, metrics-based heuristics are described. A study is conducted to validate the choice of metrics.

An approach to detect design defects by applying heuristics on design metrics has been presented by Marinescu [Mar04]. The approach provides an overall goal-oriented framework that can allow the definition of Detection Strategies. A Detection Strategy includes problem detection, meaningful filtering of the results, and composition of obtained results. Marinescu states that the low-level metrics definitions for problem detection are imprecise, which raise the issues of interpretation of the obtained metrics measurements. A pattern-like description of design defects and a systematic description of detection metrics is developed for each design flaw. For data filtering purposes, both semantic and statistical filters are used to refine the data set obtained from the application of metrics on programs. This refinement allows to obtain the results most relevant to the applied Detection Strategy. Detection Strategies are applied to search for AntiPatterns and code smells.

Detection Strategies have been enhanced by their definition in a pattern-like format as used for diseases in medical books by Trifu *et al.* [TM05]. It is stated that different code smells in code occur together because they emanate from an underlying common design defect. Therefore, the authors define these defects in a common format whereby a description is provided for a design defect, its associated code smells are listed, and the corresponding diagnosis is described for the

design defect. This format ameliorates the detection of code smells by performing a transition from their symptomatic description to a more explicit correlation amongst different code smells, and between code smells and design defects. The diagnosis suggests the application of the appropriate refactoring proposed for code smells to remove them from code.

Moha *et al.* stated that Detection Strategies are insufficient to express structural and semantical properties of design defects [MGMD08]. This insufficiency results from the limitation of the metrics to describe structural and semantical characteristics of design defects. For this purpose, a domain-specific language is described to specify defects at domain level. The language permits automatic generation of detection algorithms from defect specifications. Key concepts are extracted from literature. Key concepts correspond to the measurable properties of design problems. These include metrics-based heuristics, and lexical and structural properties. Rules describe code smells in the form of measurable properties, and AntiPatterns as a combination of code smells. These rules are used to generate automatic detection algorithms for high-level design defects.

Sahraoui *et al.* [SGM00] explored the use of object-oriented metrics as indicators to automatically detect symptomatic situations. A symptomatic situation is a structure in design or code whose metrics values indicate a poor design quality and where a particular transformation can be applied to improve the quality. Rules for the detection of a symptomatic situation are defined using quality estimation models. These models are built from the empirical results of the studies on system quality and are derived using machine learning algorithms. More concretely, a rule defines the relation of cause and effect between combinations of metric values and quality characteristics such as maintainability. The transformations suggested are similar to refactorings.

2.2.2 Query-based Design Defect Detection

Ciupke [Ciu99] proposes a technique for analyzing code, specifying frequent design problems as Prolog queries, and locating the occurrences of these problems in a model derived from the source code. The majority of detected anomalies are simple ones, i.e., simple conditions with fixed threshold values such as "the depth of inheritance tree must not exceed 6 levels".

Queries based on logic meta-programming are defined by [TM03] to detect the refactoring opportunities present in code in the form of code smells. SOUL meta-programming language manipulates Smalltalk artifacts *i.e.,* classes, methods, variables, inheritance relationships, etc. A predicate library consisting of detection rules is defined on top of SOUL to detect the code smells proposed by Fowler *et. al* in order to perform refactoring activity.

2.2.3 Design Defects Taxonomy

An effort to formalize the copious design defects (code smells, anti patterns, and design pattern defects) presented in the literature has been introduced by Moha *et al.* [MlHG06] to remove ambiguities in the interpretation of the design defects. The work actually takes different design defects as input. A tree-based taxonomy for the design defects helps classifying them in various categories and a meta-model represents all these defects in a formal, UML-type description for their automatic detection and correction. This work targets the detection and corrections of the existing set of design problems.

2.2.4 Visualizations for Design Defect Detection

Visualization techniques have been presented that aid in the detection of code smells and design defects in code [DSP08, LD03, PGN08]. Visualization techniques provide a semi-automatic approach to get a first-hand idea of the design defects present in code without delving into code-level details. But the human expertise is still required to validate the results.

The Moose reengineering environment provides a suite of tools and techniques for reverse engineering object-oriented applications [DGN05, DLT01]. These tools combine metrics and visualizations to help visualize and understand internals of object-oriented systems. Polymetric views provided by Moose is a two-dimensional visualization of nodes and edges. Polymetric views can be generated for different purposes: coarse-grained views to understand the overall system properties and fine-grained views to understand the internal of a class [LD03]. Although Polymet-

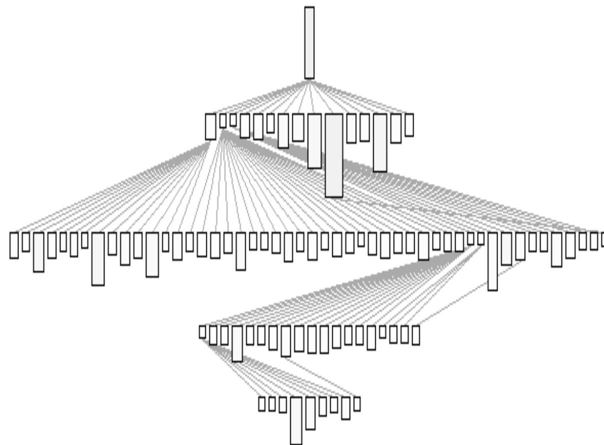

Figure 2.1: Polymetric Views: Good Design

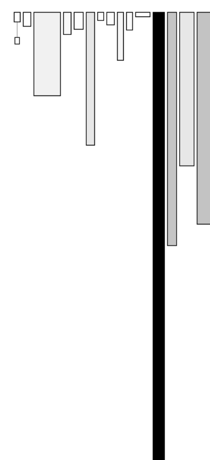

Figure 2.2: Polymetric Views: Design with Defects

ric views in Moose are described for understanding object-oriented applications, in doing so they also help identify certain design defects. In Polymetric views, visu-

alization and metrics are combined in a way that classes with anomalous metrics attributes are marked by their color or size [LD03]. While representing the overall system properties with the coarse-grained views provided by Polymetric views, nodes represent classes, while the edges represent the inheritance relationships between them. The size of the nodes reflects the number of attributes (width) and the number of methods (height) of the classes, while the color tone represents the number of lines of code of the class.

Figure 2.1 and Figure 2.2 demonstrate the difference between the visualization for a good object-oriented design and an object-oriented software demonstrating design defects, respectively. The good design is indicated by the well-decomposed classes and a good number of inheritance links. The presence of the design defects in Figure 2.2 are illustrated by the classes containing lots of methods (height of the rectangle), large amount of code (color tone), and absence of inheritance links amongst classes. However, as shall later show in this thesis that only those design defects appearing in POC can be identified with Polymetric Views that exhibit similar symptoms as the existing design defects. For different symptoms, we shall need a different approach to detect them.

Van Emden and Moonen [vM02] present the visualization of code smells in Java code. jCOSMO tool is proposed that allows the detection and visualization of the code smells proposed by Fowler *et al*. The visualization is provided in the form of graphs using RIGI [TWSM94]. Parnin *et al.* [PGN08] provide an approach to visualize the existing set of code smells present in code. For each code smell, a simple visualization is provided with the aid of colors.

SemmleCode [Sem07] allows to detect dependencies and code smells. The tool defines a query language for the detection of code smells using metrics.

However, Van Emden *et al.*, Parnin *et al.*, and SemmleCode propose approaches that are focused on the identification of existing set of code smells. These approaches are not apt to identify some of the code smells in POC that result in code scattering.

2.2.5 Other Tools

In addition to the techniques for the detection of design defects and code smells, there are various tools that are developed to help detect bug patterns, violations of coding standards, and syntax errors in code. Generally, these tools focus on improving code quality from a technical perspective. The fewer bugs there are present in a piece of code, the higher the quality of that code [vM02]. The most well-known include the C analyzer LINT [Joh77]. LINT checks for type violations, portability problems and other anomalies such as flawed pointer arithmetic, memory (de)allocation, null references, array bounds errors, etc.

PMD [PMD02], FxCop [FxC06], and CheckStyle [Che04] are some of the commercial tools that are available for searching for technical quality problems such as violations of code conventions in software. FxCop and PMD allow developers to write new custom rules for the detection of problems in code. NDe-

pend [NDe05] provides a useful tool to calculate various quality metrics for .NET programs and information for type-based dependencies.

CROCOPAT [BNL05] tool allows several structural analyses in object-oriented systems. The tool allows to search for various problems such as the detection of design patterns, cyclic code dependencies, duplicate code, and dead code.

2.2.6 Limitations of Design Defects Detection

All the heuristic-based approaches for design defect detection provide an overall framework for the detection of the existing design flaws. These approaches rely on software quality metrics and the textual description of the existing design defects and code smells. Their relevance with the work presented in this thesis lies in the fact that the code smells that we bring forth and their proposed detection detection techniques can be formally described with these approaches.

Other query-based and visualization tools that aim for the design defect detection for a transition towards a superior object-oriented design focus to search for existing set of design defects, such as AntiPatterns and code smells. These tools, with the existing set of detection techniques, cannot be directly used to detect the code smells appearing in POC that result in scattered code. The limitations of existing tools and techniques lies in the fact that the existing techniques rely on code quality metrics for code smell detection. However, scattered code cannot be detected with code quality metrics as it requires a structural analysis for the scattered code. Moreover, all the locations where scattered code exists should be detected. Thus, we believe that a new set of techniques are required to detect for scattered code. Hence, when we describe the new set of code smells that appear in the absence of design, we also provide a some techniques that help detect them. Our proposal for scattered code detection will help the existing tool developers to integrate the proposed detection techniques in their respective tools.

2.3 Object Identification and Class Restructuring

While the design defects and code smells should be described and detected in code, the ultimate objective for their description remains their removal from the code. Their removal requires the identification of the modifications that are needed to be applied to transform programs so that they represent a superior object-oriented design.

Several research works aim to identify objects in procedural programs to transform them into an improved object-oriented design. Moreover, there are various studies that propose to restructure object-oriented software into an improved design by removing design defects and code smells. In the following sections, various object identification approaches in procedural programs are discussed. Afterwards, we present approaches that propose to restructure object-oriented programs; specifically, we mention those approaches that rely on the usage of Formal Concept

Analysis (FCA) [GW99].

2.3.1 Object Identification in Procedural Code

Graph-based Object Identification. Several work attempted to transform procedural code to object-oriented one [CB91, CCM96, CLLF99, LFGP97]. Newcomb *et al.* [NK95] proposed a Hierarchical Object-Oriented State-Machine Model that is between conventional object-oriented modeling languages, state-based reactive specification systems, and event-driven programming models. COBOL records are mapped to classes and each procedure is mapped to a state machine associated to a method. Several refactorings and transformations are applied to abstract and merge the resulting methods.

De Lucia *et al.* [CLLF99, LFGP97] describe the Ercole approach for migrating programs written in RPG, the business application programming language, to object-oriented programs. Among the different steps of the approach, one is abstracting an object-oriented model that is centered around the persistent data stores. Subroutines and groups of call-related subroutines are then candidate methods.

Cimitile *et al.* [CLLF99] found the identification of objects on the optimization of certain object-oriented design metrics. The method identify coarse-grained objects whose state is implemented by persistent data stores. First, a static analysis of the source code is performed to identify the persistent data stores used and their structure. The results are then refined through an analysis of the synonyms and a concept assignment process. Synonymous files are grouped to form one object. The concept assignment process is performed to associate a concept of the application domain to each of the identified objects and to aggregate data stores corresponding to the same element of the application domain. Object operations are searched for at the program level first: object-oriented design metrics are used to assign programs as object operations. In particular, the assignment is made while trying to minimize the coupling between the objects. Measures of the coupling between programs and persistent data stores are computed based on the accesses that programs make to the data stores.

An object-identification algorithm is proposed which treats coincidental and spurious connections with a statistical technique [CCM96]. The algorithm exploits an interconnection graph, called a *variable-reference* graph, and the objects are identified by looking for mainly internally connected sub-graphs, which the algorithm attempts to identify through an iterative process. At each step, every routine in the system is associated with an index that measures the variation in the internal connectivity of the graph that results in the use of the routine to generate a new cluster. A filtering function is then used to discriminate, on the basis of the associated index, whether a function has to be used to generate a cluster, or it is likely to introduce undesired connections.

Gall and Klösche propose a distinctive approach that is not only based on the information derivable from the source code, but it also integrates domain and application-specific knowledge to achieve the identification of application-semantic

objects [GK95]. In the approach, object candidates are derived from the code by the analysis of data store entities (persistent data) and non-data store entities (non-persistent data). This analysis is also used to detect any functional relationships amongst various entities. This model is called *RooAM* (Reverse generated object-oriented Application Model). An independently developed object-oriented application model called *FooAM* is generated for the same application from specifications. A FooAM model is then used to direct the object identification process and complete the elements and resolve any ambiguities in the RooAM model.

Evidence-driven Object Identification. Evidence-driven identification is based on an assumption that the user can discover a better object-oriented design once he or she has global view of the different design alternatives [KP99]. The process starts by identifying initial set of candidate classes and set of candidate methods. However, the methods attributed to class may be in conflict. Hence, the next step is to assign conflicting methods to one of the candidate classes. The idea is to construct for each method global evidence table per candidate class. Each column contains a property that holds for the method and associated class based on their structural relationship. Weights are assigned to each column according to user priority. A method is more suitable for a candidate class A than B, if its sum of weights in evidence table of A is more than that in evidence table for B.

FCA-based Object Identification. Concept formation methods have been applied for object identification in procedural code. Sahraoui *et al.* in [SMLD97] proposed an approach for identifying objects in procedural code. The approach combines metrics calculation with several FCA-based analysis steps for class identification and further graph-based reasoning to detect method associations for newly identified classes. In the first phase, groups of global variables and functions that operate upon them are identified. The usage of variables by methods is further divided into modification, read, and predicate mode. The predicate model indicates if the variable is used to control execution of a function. The control execution information of a module is used to identify candidate objects through concept lattice. In the second phase, candidate objects are further refined by grouping those candidate objects that share common variables. Concept lattices for candidate objects and their associated variables is constructed to merge similar objects. In the third phase, behavior is attached to the identified objects. For this purpose, reference (access) and modification relationships of functions and variables are computed. Modification relationship is given preference over reference (access) relationship. As we shall demonstrate later, our approach for restructuring class in POC carries forward this approach in that we generate more than one FCA views to extract useful class hierarchies from POC classes.

Arie van Deursen proposes to use FCA and clustering to use legacy data structures as a starting point to object identification [vDK99b]. A legacy COBOL system is studied for object identification. The authors apply cluster and concept

analysis for object identification in the program and provide an analysis for the strengths and weakness of the two techniques vis-à-vis object identification. The authors have remarked that concept analysis is more apt for object identification because it produces the same candidate objects irrespective of the order in which records are chosen. In clustering, the appearance of an item in a give cluster depends upon the already computed clusters.

An approach to transform a COBOL legacy system to a distributed component system is proposed by Canfora *et al.* [CCDD99]. The overall purpose is to reduce the complexity of the lattices through the subgraph identification by the application of an eclectic approach. A concept lattice is generated for programs and files (database tables) used by these programs. Rules are defined for removing irrelevant files and programs. For example, files that do not implement domain objects and programs that access one file are removed. This reduction of context information for lattices reduces the complexity of the resulting lattice and aids in simplifying object identification. The focus remains the decomposition of COBOL programs and data into meaningful components.

Clustering-based Object Identification. Object identification can be seen as grouping problem. Different clustering techniques [Wig97] such as Optimization Algorithms or Hierarchical Algorithms could be used to detect objects. Sahraoui *et al.* [SVKS02] applied genetic algorithm and conceptual clustering algorithm to detect objects in procedural code. Population consists of chromosomes. A chromosome consists of groups of variables. Initially a random set of chromosomes are produced *i.e.,* random groups of variables. The method selects the best chromosomes by applying objective function to chromosome sets. Then, mutate and crossover operators are applied on each of the selected chromosome pairs to obtain new chromosomes. The new chromosomes are found to have better groups of variables compared their previous generations.

Object Oriented Idioms Found in C. Procedural code such as C implements inheritance in variety of ways. Some interesting patterns have been identified that are used in C programs to simulate object oriented features such as inheritance [Sif98, Har00]. The simplest way in which inheritance is implemented is through redundant declarations [Sif98]. The redundant declarations are shown below with a sample declaration:

```
typedef struct { int x,y; } Point;
typedef struct { int x,y; Color c; } ColorPoint;
```

The example above describes ColorPoint that is a subclass of another class Point because they have similar data structures, x and y, in their declaration.

2.3.2 Object-Oriented Restructuring

Moha [Moh08] suggests that the correction of defects in object-oriented design consists of three steps, namely, identification of modifications, application of the modifications, and testing the resultant system. The modifications, or commonly refactorings, that are necessary to improve object-oriented design are defined by Opdyke [Opd92] and Fowler [FBB$^+$99]. We believe that the modifications to be applied to remove design defects and code smells are adequately described by existing research. Testing the results of refactorings is a manual task performed by developers.

Hence, the most important task in object-oriented restructuring is the identification of an overall roadmap to suggest and pursue refactoring activity to remove design defects. This roadmap should extract help extract an overall object-oriented design from POC programs.

First we provide an introduction of the proposed refactorings. Then, we list pertinent approaches related to finding an overall roamap to reengineer and restructure object-oriented code to an improved design.

Refactorings. Opdyke first introduced the concept of refactorings to improve program design [Opd92]. The purpose of these refactorings is to change the structure of a program in a useful way, and in doing so refactorings must preserve program behavior. The proposed refactorings consist of two groups: low-level refactorings and composed refactorings. Low-level refactorings include creating an empty class and creating a new member attribute in a class. Low-level refactorings are defined to show that they preserve program behavior. It is stated that all the composed refactorings will preserve program behavior that are composed of a series of low-level refactorings. Opdyke has defined 26 low level refactorings.

Opdyke has also contributed to Martin Fowler's book [FBB$^+$99] on code smells and refactorings. The book provides a comprehensive list of refactorings to improve object-oriented design. Here, we present three examples of refactorings proposed in the book for illustration purposes.

- *Move Method.* The refactoring proposes to move a method of a Class A to another Class B. The purpose is to move methods to the classes where these methods are more frequently used.

- *Extract Superclass* The purpose of this refactoring is to move common features in sibling classes A and B in a common superclass C. Common features such as duplicate code are pulled up into the newly created class C. Classes A and B then inherit from the superclass C.

- *Extract Class.* This refactoring consists of creating a new class B and displacing all the pertinent attributes and methods from Class A to the newly created Class B. This refactoring is used in cases where a single class en-

capsulates two different concepts. After the refactoring, each class contains functionality pertaining to its purpose.

A meta-model based refactoring engine is proposed by Tichelaar *et al.* [TDDN00]. The purpose of the refactoring engine is to perform the proposed refactorings in a language-independent manner. The meta-model is called FAMIX and it consist of object-oriented entities such as methods, classes, and attributes. Based on the list of primitive refactorings, the authors derive a common meta-model. The feasibility of the application of approach is studied for Java and Smalltalk programs.

FCA-based Class Hierarchy Reengineering. FCA is proposed for class hierarchy reengineering by Snelting and Tip [ST98]. The authors proposed an FCA-based method for adapting a class hierarchy to a specific usage thereof. It comprises a study of the way class members are used in the client code of a set of applications. The study enables the identification of anomalies in the design of class hierarchies, e.g., class members that are redundant or that can be moved into a derived class. However, the work does not address the presence of high-level object-oriented defects and their correction.

FCA-based Object-Oriented Restructuring. Moha *et al.* [MHVG08] proposed an approach to use Relational Concept Analysis (RCA) to suggest appropriate refactorings to correct certain design defects. The approach suggests to make classes more cohesive and less coupled by removing Blob anti-pattern. RCA lattices are used to refactor to remove Blob AntiPattern. For doing this, an RCA context is constructed where methods are considered as objects and method calls and class attributes are considered as attributes of the context. Lattices are then scaled to obtain useful information.

FCA-based Class Hierarchy Understanding. FCA-based understanding of class structures is introduced in [ADN05a]. First, the authors identify pattern of views based on FCA to understand the access of class attributes and method usage for existing class hierarchies. Various collaborations are defined through the analysis of class state and behavior, and interactions between the state and behavior. Based on the groupings of these interactions, views provide internals of class interaction patterns. These views include State Usage, External/Internal Calls, and Behavioral Skeleton. State usage demonstrates the usage patterns of class attributes by methods of the class. External/Internal calls reveal the overall shape of the class in terms of its internal reuse of functionality by depicting methods that are not called from inside their own class and that the methods only act as accessors to class attributes. Finally, behavioral skeleton demonstrates the methods that do not use class data and methods. These views are useful in understanding the internals of object-oriented classes.

Dekel uses FCA to visualize the structure of Java classes and to select an effective order for reading the methods [DG03]. Method call graphs is superimposed

onto the concept lattice to obtain an embedded call-graph, which provides a detailed visualization of the interaction within a class.

2.3.3 Limitations of object-oriented Restructuring

Limitations of Procedural Object Identification. There are copious approaches on object identification in procedural code. However, these approaches cannot be applied to POC for two reasons. First, these approaches do not include the identification of object-oriented constructs such as method calls, composition, and association relationships while searching for object-oriented classes. The integration of these constructs in the restructuring approach is necessary to exploit the existing underlying object-oriented constructs while constructing new class hierarchies. Second, these approaches do not take into account the POC code smells, which can also provide certain hints for the presence of class hierarchies, as we shall show later.

Limitations of FCA-based object-oriented Reengineering. The proposals for understanding class hierarchies through the use of FCA search to understand class hierarchies and remove petty design anomalies present in class hierarchies such as misplaced attributes in subclasses that should be moved up in the class hierarchy. However, they do not serve the purpose of removing high-level design defects from object-oriented code, especially, when the program demonstrates the lack of the overall design.

RCA-based technique to remove AntiPatterns from code produce huge lattices for large systems that obstruct the task of the inference of useful object-oriented abstractions from code. The solution that we shall propose looks to decompose lattices to represent each cohesive group of attributes and methods.

2.4 Identification of Crosscutting Concerns

In addition to the design problems appearing due to deviations of object-oriented norms, it has been reported that certain features, or more specifically concerns, can not be encapsulated into the existing object-oriented abstractions and result in crosscutting concerns [KLM+97]. Aspect mining techniques search for scattered code related to the absence of Aspect-Oriented Programming constructs. Below, first we provide a brief introduction of crosscutting concerns. Afterwards, we provide a detailed listing of tools and techniques for the identification of crosscutting concerns. The purpose is to highlight the absence of literature on the detection of scattered code resulting from the absence of object-oriented design.

2.4.1 Crosscutting Concerns

A commonly used definition of a concern is given by IEEE 1471: a concern is an interest that pertains to the system's development, its operations and any

other aspects that are critical or otherwise important to one or more stakeholders [FECA05]. Software is a collection of various concerns. There are certain concerns which cannot be cleanly encapsulated in the current decomposition units offered by programming languages, including object-oriented paradigm [Kic96]. When such concerns are implemented in a system, they result in *Crosscutting Concerns*. Crosscutting concerns exhibit symptoms of code duplication, and scattering and tangling of code. In turn, they affect the modularity of a system because changes to these crosscutting concerns are not bounded to one module but spread to various modules in the system.

```
class Point { int x, y;
public int getX() { return x; }

public int getY() { return y; }

public void setX(int x) {
     this.x = x;
     Display.update();
}

public void setY(int y) {
     this.y = y;
     Display.update();
}

public void moveBy(int dx, int dy) {
     x += dx;
     y += dy;
     Display.update();
} }
```

Listing 2.1: Non-AOP Code

Listing 2.1 depicts an example of a class defining the state for a class *Point* and the operations on Point coordinates. The example is a simplified version of the example presented by Kiczales ad Mezini [KM05]. Every time x or y coordinates of the class Point are changed or the point is moved, the display is updated through the *update* operation. The calls to the update method of Display class are scattered in the methods of the class Point. Moreover, all the classes that inherit from the point class inherit the logic of update and hence the scattered logic is inherited too.

```
class Point {

int x, y;

public int getX() { return x; }
```

```
public int getY() { return y; }

public void setX(int x) { this.x = x; }

public void setY(int y) { this.y = y; }

public void moveBy(int dx, int dy)

{
    x += dx; y += dy;
}
}

aspect UpdateSignaling {
pointcut change(): execution(void
Point.setX(int)) ||
execution(void Point.setY(int)) ||
execution(void Point.moveBy(int, int));
after() returning: change()
{
    Display.update();
} }
```

Listing 2.2: AOP Version

Listing 2.2 shows Aspect-Oriented Programming [KLM$^+$97] (AOP) version of the Point class. The update logic is now refactored to the aspect *UpdateSignaling*. The pointcut *change* captures the execution of methods setX, setY, and moveBy. Each time the pointcut captures the execution of one of these methods, it executes the update operation after returning from the methods setX, setY, and moveBy. The update logic is now well-encapsulated in a single aspect. Hence, using AOP, these crosscutting concerns can be cleanly separated from the base code, which becomes oblivious of the update operations [FF00].

2.4.2 Aspect Mining

AOP and its relevant concepts are rather new and industrial systems have been using object-oriented paradigm as the state of the art technique to build modular systems. So, there is a need to search for crosscutting concerns in existing systems so that scattered code can be refactored into aspects [HRB$^+$06]. Manual identification of crosscutting concerns in the existing system is a difficult and error-prone process due to large size, complexity, and obsolete documentation and knowledge of existing systems. Hence, tools and techniques are essentiel to assist the discovery of crosscutting concerns.

Aspect mining refers to the identification and analysis of non-localized crosscutting concerns throughout an existing legacy software system [KMT07]. The detected concerns can be re-implemented as separate aspects, thereby improving maintainability and extensibility as well as reducing complexity. The overall process of aspect mining and refactoring has been described in Figure 2.3 (taken from [KMT07]).

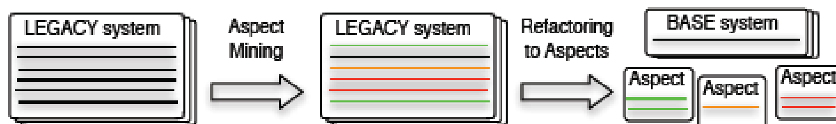

Figure 2.3: Aspect Mining and Refactoring [KMT07]

The origins of aspect mining can be traced back to the concept assignment problem, i.e., the problem of discovering domain concepts and assigning them to their realizations within a specific program [BMW94]. In aspect mining those concerns (concepts) are discovered whose realization in a given program cuts across modular units. Hence, the general hypothesis for aspect mining, be it in requirement documents, program behavior, or code, is that the manifestation of aspectual opportunities is redundancy. This redundancy appears as scattered artifacts in documents or code. Hence the basic assumption of aspect mining techniques is to find the scattering of artifacts. These artifacts may be words in documents, recurrent code elements, method calls, or execution patterns. Therefore, aspect mining techniques are distinguished as belonging to three categories: early discovery techniques, dedicated code browsers, and automated aspect mining techniques [MKK08]. Figure 2.4 demonstrate an overview of existing aspect mining tools and techniques [KMT07]. These tools and techniques are elaborated in the following sections.

Aspect mining involves the search for source code elements belonging to the implementation of a crosscutting concern. These source code elements are called *candidate seeds*, which can be turned into confirmed seeds if accepted by a human expert, or non-seeds if rejected. A non-seed appearing in the results of aspect mining tools is referred to as a false positives. Too many false-positives produced by an aspect mining tool requires a major human effort because of the need to study and reject all the false-positives reported by the tool. Lesser number of false-positives is also a prerequisite for a successful aspect mining and refactoring activity. Hence, aspect mining tools and techniques strive to reduce the number of false-positives identified in their results.

Starting from the specification documents, it has been argued that the aspect-oriented paradigm should be adopted from the early stages of software development such as domain analysis and requirements engineering [SCRR05].

In the following sections, various aspect mining techniques for searching miss-

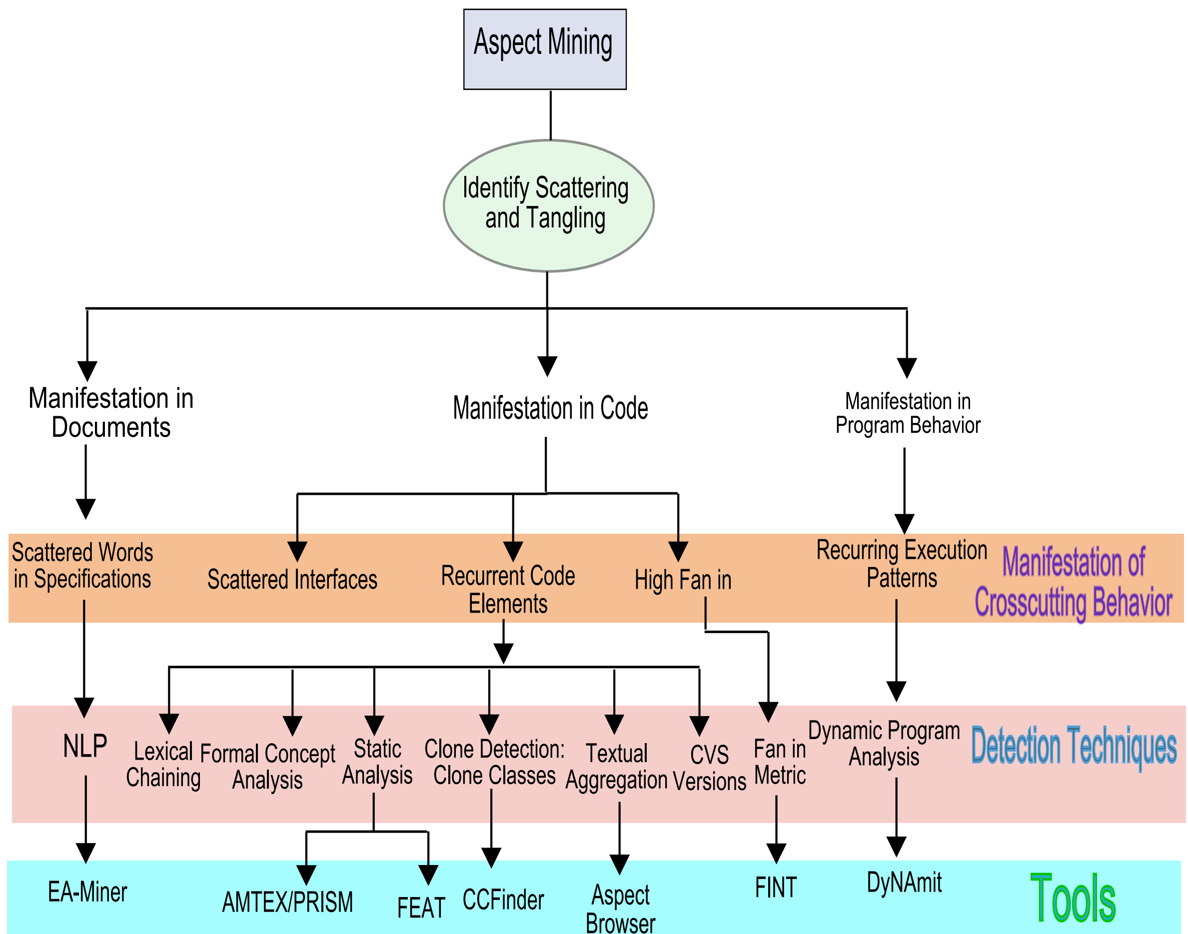

Figure 2.4: Aspect Mining Tools and Techniques

ing aspects in application code are discussed. For this purpose, we divide these techniques into two groups: dedicated code browsers and automated techniques. Dedicated code browsers need user input for searching for useful candidates seeds. Automated techniques generate a set of candidates, which are further analyzed manually to determine real crosscutting concerns. We discuss members of each type of aspect mining techniques below.

Dedicated Code Browsers. Dedicated code browsers rely on search patterns provided by aspect miner to identify aspects. Thus, they are also termed as query-based or explorative techniques [MDM07]. These techniques guide an aspect miner who has an idea of the aspects in the code and searches for crosscutting concerns providing lexical or type patterns to code browsers. Source code locations that match the pattern correspond to crosscutting concern seeds, which can

subsequently be analyzed in detail using features provided by the code browsers.

Aspect Browser is one of the first aspect mining approaches [GKY00]. It uses lexical pattern matching for querying the code, and a map metaphor for visualizing the results. It extracts fragments of identifier names from source code according to a programmer specified naming convention.

The AspectMining Tool (AMT) extends the lexical search from the Aspect Browser with structural search for usage of types within a given piece of code [HK01]. The tool displays the query results related to crosscutting concerns in a Seesoft-type view as highlighted strips in enclosed regions representing modules (e.g., compilation units) of the system [ESEE92].

AMTEX is an AMT extension that provides support for quantifying the characterization of particular aspects. AMTEX, in turn, has evolved into PRISM, a tool supporting identification activities by means of lexical and type-based patterns called fingerprints [ZJ04]. A fingerprint can be defined, for example, as any method, of which the name starts with a given word. An aspect miner defining fingerprints is assisted by so-called advisors. PRISM currently provides a ranking advisor which reports the most frequently used types across methods.

The Feature Exploration and Analysis Tool (FEAT) is an Eclipse plug-in aiming at locating, describing, and analyzing concerns in source code [RM07]. It is based on concern graphs, which represent the elements of a concern and their relationships. A FEAT session starts with an element known to be a concern seed, and FEAT allows the user to query relations, such as direct call relations, between the seed and other elements in the program. The results of the query that are considered relevant by the user to the implementation of a (crosscutting) concern can be added to the representation of the concern.

Various query-based tools (the Aspect Browser, AMT, and FEAT) have been compared in a recent study [FECA05]. This study shows that the queries and patterns are mostly derived from application knowledge, code reading, words from task descriptions, or names of files. As the study shows, prior knowledge of the system or known starting points strongly affect the usefulness of the outcomes of the analysis.

Automated Techniques. The second group of aspect mining approaches aim at automatically generating crosscutting concern seeds that will reduce the effort of further understanding and exploring the concern. The approaches in this category can be described as automated or generative techniques and will typically provide the input for the dedicated code browsers.

Many automated approaches use program analysis techniques to look for symptoms of code scattering and tangling and identify code elements exhibiting these symptoms that can act as candidate aspect seeds. Shepherd *et al.* [SGP04] use clone detection based on program dependence graphs and the comparison of individual statements abstract syntax trees for mining aspects in Java source code. Three clone detection tools, implementing matching on tokens, abstract syntax trees, and

on program dependence graphs, respectively, have been evaluated on an industrial C component [vEBvDT05]. The starting point were four dedicated crosscutting concerns that were manually identified and annotated in the code beforehand. The evaluation assesses the suitability of clone detection for identifying these concerns automatically by measuring the coverage of the annotated concerns by detected clones.

Code clones in object-oriented systems would typically be refactored through method extraction [FBB+99] that results in scattered calls to the extracted method. Fan-in analysis looks for the concerns implemented by these scattered calls, which could be further refactored into aspect advice [MDM07]. Unique methods searches for crosscutting concerns by identifying methods with higher number of calls [GK05]. This technique concludes that crosscutting concerns provide "services" to other concerns and their associated methods, when called, do not return data. In their more recent work, Breu and Zimmermann searched for concerns by analyzing the changes in the values of the fan-in metric between different versions of the system under investigation [BZ06]. The technique they propose examines the version history for insertions of method calls. Similar to fan-in analysis, a reported seed consists of a set of one or more methods with same call site locations.

Dynamic analysis has been considered for aspect identification by examining execution traces for recurring execution patterns [BK04] and by applying formal concept analysis to associate method executions to traces specific to documentation-derived use-case scenarios [TC04a]. Particularly challenging for dynamic analysis techniques is to exercise all functionality in the system that could lead to aspect candidates. The execution of the functionality for dynamic analysis implies that a preliminary activity is needed in which use-case scenarios are defined for the system under investigation. The first of the two dynamic techniques has been adapted recently to static analysis to search for recurring execution patterns in control flow graphs [Kri06].

Formal concept analysis has also been applied in an identifier analysis that groups programming elements based on their names [TM04]. This analysis starts from the assumption that naming conventions in programs can be used to detect the scattered elements of a concern.

The suitability of refactoring certain interfaces implemented by a class has been investigated through a number of indicators like the naming pattern used by the interface definition, the coupling between the methods of the implementing class and the methods declared by the interface, or the package location of the interface and its implementing class [TC04b].

Aspect Identification Case Studies. Various aspect mining case studies have been performed [CMM+05, RURD07, SPPCC05]. These case studies are geared towards the analysis of combination of aspect mining techniques and comparison of the precision of their seed identification. A first attempt was made in [CMM+05] to establish a common benchmark for the development of aspect mining techniques

and to report results for various aspect mining techniques. This work combines results of dynamic analysis [TC04a], fan-in technique [MDM07], and identifier analysis [TM04] approaches. This work is further studied and results are analyzed in [RURD07] for the same software system.

Aspect mining tools are only tested on programs exhibiting good object-oriented design. Likewise, the case study software chosen, such as JHotDraw [JHo], demonstrate a good overall object-oriented design. Hence, aspect mining tools results do not report the existence of false-positive results occurring due to the absence of object-oriented design.

2.4.3 Concern Quantification

Apart from aspect mining tools and techniques, there are techniques that target the quantification of the scattering of code elements pertaining to various concerns. Concern identification and interaction through manual feature selection tool and a first set of metrics for feature scattering have been presented in [LM99]. Primitive metrics such as spread, tangle, and density are provided for various features present in a program. These features are manually tagged by the user to understand the scattering of the code elements related to each tagged feature.

Wang *et al.* undertook a study to compute, using dynamic analysis of code, various features implemented in code and calculate the relationship of these features with program components through disparity, concentration, and dedication metrics [WGH00]. Program features are determination by their appearance in execution traces. These features are then mapped onto the file and disparity, concentration, and dedication metrics are calculated for features implemented in a program.

Eaddy *et al.* have presented a manual approach for concern identification and concern assignment and presents two concerns quantification metrics derived from [WGH00]: degree of scattering and degree of focus [EZS+08]. The proposed approach promotes manual identification of concerns and discuss crosscutting concerns arising in programs.

Garcia *et al.* have also provided a set of metrics to measure the scattering of crosscutting concerns. These metrics are called Concern Diffusion over Component (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over Lines of Code (CDLoc) [KSG+06]. As their names suggest, these metrics are used to measure the diffusion of a concern over classes, methods, and lines of code in object-oriented programs. The assignment of code artifacts to concerns is manual.

An experience of marking concerns in two programs with a tool, SPOTLight, has been presented in [RBC05]. This tool serves to associate code snippets with various concerns and concern markings of the two developers have been compared by analyzing the overlap of the lines of code of their respective concerns. The work identifies useful guidelines for concern identification.

A terminology and formalism along with a set of criteria for the comparison, evaluation, and definition of existing concern metrics is proposed in [FSG+08].

The paper proposes guidance for consistent concern measurement in an abstract manner by providing a survey of currently proposed metrics.

2.4.4 Aspect Refactoring

Aspect refactoring refers to the extraction of the code related to identified crosscutting concerns into AOP language constructs [HRB+06, Lad03, Mar04]. Aspect refactoring works presume that prior to aspect refactoring, crosscutting code is already identified. In order to enable the refactoring of crosscutting code into aspects, sometimes object-oriented refactorings are applied to make joinpoints *i.e.,* points in code where aspects are applied, more explicit in code [HRB+06]. It is also suggested that object-oriented refactoring should be more aspect-aware because aspects directly depend upon the joinpoints present in base programs [HOU].

Monteiro *et al.* [Mon05] mention the prerequisite of a good object-oriented design before the application of aspect refactoring. In summary, aspect refactoring works set the precondition of good object-oriented design for applying aspect refactoring. Works on aspect refactoring do not report the absence of object-oriented design and resulting code scattering that may appear. In cases where there are anomalies, object-oriented refactorings are applied to prepare the programs for aspect refactoring.

2.4.5 Limitations in Crosscutting Concerns Identification

The work in the domain of aspect identification, aspect refactoring, and concern identification only strives to search for the scattered code from aspect identification dimension and their eventual refactoring. In this thesis, we demonstrate that scattered can also appear due to the absence of classes for domain entities. The scattered code appearing from the absence of object-oriented design is never reported in the literature. We also demonstrate that aspect mining techniques can equally be used to detect some of the code smells related to the absence of object-oriented design. On the one hand, the application of aspect mining techniques allows to detect scattered code related to the absence of object-oriented design and, on the other hand, it requires that the scattered code related to the absence of design be removed from the aspect mining results for better accuracy of aspect mining tools. We shall describe an approach for the classification of scattered code appearing in non-AOP object-oriented programs demonstrating POC design defects.

2.5 Discussion

This chapter discussed various proposals related to the design problems in object-oriented programs and their detection techniques. We also presented diverse approaches that help restructure procedural and object-oriented code into an improve object-oriented design. In addition, we also described extensive list of techniques

aspect mining and concern identification. In summary, we list the limitations of the described proposals regarding our problem statement.

2.5.1 Design Defects and Code Smells

Code smells described by Fowler [FBB+99] only describe localized design anomalies in object-oriented programs that do not occur because of the absence of design. Existing repository of design defects, or more specifically AntiPatterns, describe high-level design problems that arise from deviations of design norms. These high-level design problems are described in an abstract manner with textual description of code problems. We believe that AntiPatterns do not describe all forms of code-level manifestations of a degenerated object-oriented software. Hence, developers and reengineers can always encounter and report a different manifestation of code-level problems related to similar higher-level design defects. We shall present a list of the design defects and code smells that we observed in software lacking a proper object-oriented design.

The existing techniques for the detection of the existing design problems are insufficient because they search the existing symptoms of design problems in code. Any new symptoms would require the definition of new techniques for their detection. Some of the code patterns that appear in the absence of object-oriented design produce scattered code. The existing set of proposals for design defect detection in object-oriented do not serve to detect the scattered code that results from the absence of object-oriented design. Hence, techniques are required to be defined to search for the scattered code.

2.5.2 Object-Oriented Restructuring

There is extensive literature on object identification in procedural code. Moreover, there are several proposals for restructuring of object-oriented programs into an improve object-oriented design. While searching for an appropriate technique to identify refactoring opportunities in POC, we found the following limitations of the existing proposals.

- Object identification in procedural code does not include object-oriented constructs such as method calls, composition, and association relationships while searching for object-oriented abstractions.
- FCA-based techniques for class hierarchy reengineering and understanding attempt to locate anomalies in classes and class hierarchies. These do not attempt to resolve high-level design defects appearing due to the lack of object-oriented design.
- FCA-based object-oriented restructuring involves producing a single lattice for a system. This obstructs the analysis of huge systems. Moreover, class hierarchies for domain entities cannot be extracted with the proposed technique.

- The approaches for FCA-based restructuring do not take into account the code smells that appear in the absence of object-oriented design while searching for classes that can better encapsulate domain entities.

2.5.3 Aspect Mining

Aspect mining strives to search for scattered code related to crosscutting concerns present in code. However, the proposed techniques aim to search for scattered code resulting from the absence of aspects. We believe that the scattered code related to the absence of classes for domain entities can also appear in the results of aspect mining tools. This scattered code introduces false-positives in aspect mining results, thus reducing the accuracy of the results obtained. The scattered code related to the absence of classes for domain entity is never reported in the existing literature on aspect mining. Moreover, concern identification and quantification tools and techniques do not address the question of scattered code occurring due to diverse reasons and its classification.

2.5.4 Proposed Solution

In the next chapters, we describe our contributions that fill up the aforementioned gaps in the existing research. We summarize here our research contribution:

- As the current state of the art does not mention a description of the higher-level design defects and their code-level manifestation occurring in the absence of object-oriented design, we present the design defects and code smells appearing in POC. Their description will help better identify and remove them from code to improve the overall object-oriented design. We also discuss identification techniques to detect the code smells that result in scattered code. We utilize techniques to detect scattered code to detect the POC code smells result scattered in code.

- Object identification and object-oriented reengineering techniques are not adequate for the extraction of useful abstraction from POC to correct the design defects. For object identification in procedural code, cohesive groups of variables and functions are detected. However, these approaches do not take into account object-oriented constructs. Class hierarchy reengineering, provides a single lattice for the whole application that hinders the extraction of cohesive groups of methods and attributes and their hierarchical links. For the purpose of POC restructuring, we present an approach that helps restructuring POC classes. First, we identify cohesive groups of methods and attributes. To decompose lattice information, we generate three different FCA views for each of the cohesive groups to analyze these cohesive groups individually. The decomposition permits to identify class hierarchies within each of the cohesive groups of methods and attributes.

- Crosscutting concerns identification in POC results in a large number of false-positive aspect to be identified, which point to the absence of object-oriented domain entities. The existing approaches for aspect mining and concern quantification do not report the problem of false-positive aspects. We propose an approach to distinguish crosscutting concerns through structural analysis of programs and scattering metrics. The approach identifies a list of crosscutting concerns and distinguishes those that access domain entity information as related to the missing objects. Moreover, Spread-out metric quantifies the scattering of concerns on classes. Aspects demonstrate high scattering values as compared to the missing objects. The approach permits to identify and extract those crosscutting concerns that appear from the absence of object-oriented design from the list of candidate aspects for improving aspect mining results.

Chapter 3

Procedural Object-Oriented Code

While the benefits of object-oriented technology are widely recognized, the indiscriminate use of object-oriented mechanisms and weaknesses in analysis and design methods are rapidly leading to a new generation of inflexible legacy systems. [Cas98]

Le problème ne cherche personne, c'est la personne qui cherche le problème. Proverbe Camerounais

3.1 Overview

The object-oriented paradigm provides important concepts that help construct modular software systems. These concepts include inheritance, polymorphism, and composition/aggregation. They are dependent upon the application of object-oriented analysis and design before the actual programming phase so that classes and hierarchies for domain entities are identified. However, software systems lacking a good object-oriented design or in case of design erosion, the key object-oriented concepts are absent and hence software systems suffer from modularity problems. We term software systems developed using the state of the art object-oriented languages but lacking object-oriented design or demonstrating design erosion, *Procedural Object-Oriented Code* (POC). Huge classes, shallow class hierarchies and the absence of domain abstractions in code are the hallmark design defects of POC. These design defects manifest themselves in the form of various code smells in programs. A few examples of code smells are displaced methods and duplicate template code to name a few.

In the current chapter, we elaborate the design defects appearing in POC and their manifestation in the form of code smells along with examples. The description of the design defects and resulting code smells is necessary so that these can be detected and removed from the code for an improvement in software quality.

Design defects and code smells identification is essential for their removal from code. However, manual identification can be cumbersome in large-scale software. Their automated identification is essential to reduce the burden of manually searching for the POC design defects. Hence, we briefly describe various existing approaches based on code quality metrics and visualizations for the detection of the design defects and the code smells occurring in POC. We show that some of the code smells cannot be identified through the existing techniques for two reasons. First, the existing proposals apply techniques, such as metrics-based heuristics, which are not pertinent to detect the new set of code smells in POC. Second, the existing detection techniques search symptoms of the existing design defects and code smells. The new set of code smells exhibit different symptoms than the current ones: Some of the POC code smells result in scattered code. These new code smells necessitate the definition of a tool that can exploit the code scattering identification techniques to search for these code smells. Thus, we propose a techniques, which is based on the principles of the detection of scattered code to detect the POC code smells exhibiting the scattered code. The techniques proposes to detect the scattered code through identifier analysis and Fan-in metric.

This chapter is organized as follows: Section 3.2 describes the main concepts of the object-oriented paradigm to illustrate these concepts and to contrast them with their absence in POC. Section 3.3 describes POC and Section 3.4 describes the design defects and the corresponding code smells present in POC with examples. Section 3.5 describes various tools and techniques that may be useful to automate the identification of the POC design defects and code smells. Section 3.6 concludes the chapter.

3.2 Background — Object-Oriented Paradigm

Different methods have been proposed for designing and developing modular software systems, one of the recent one is the Object-Oriented paradigm [Mey88]. Object-oriented languages allow developers to define *classes* (a unit of modularity) of objects that behave in a controlled and well-defined manner. Classes represent domain entities, which are the real world entities modeled by software systems. Domain entities are normally identified during object-oriented analysis and refined during the design phase [Pre01]. For example, the domain entities for library management system would be books, shelves, and users. Likewise, the domain entities for blood plasma analysis applications would be patients, tests, quality control, tubes, reagents, etc.

Classes encapsulate data and functionality related to domain entities [Rie96]. The principle to encapsulate data and functionality together is known as data encapsulation and the objects expose their functionality through well-defined interfaces. The separation of interface and its implementation lies at the heart of encapsulation and information hiding principle [Par72]. Interface contains only those elements that a client class needs to know while hiding the implementation details behind it.

The separation of implementation details and interface constitutes loose-coupling between a provider object and its client. The client knows how to use the object without knowing how the provider object is built. Hence the provider object can evolve its private implementation without changing its interface. The changes are localized, thus supporting the concept of modular continuity [Mey88].

The concepts of abstraction and encapsulation are supported in the object-oriented paradigm by inheritance and aggregation. Inheritance — which enables parts of an existing interface to an object to be changed — enhances the potential for re-usability by abstracting commonly used attributes and functionality in top-level classes. Polymorphism is strongly connected to inheritance. Polymorphism is the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior. Some objects are made of other objects and it is called aggregation or composition.

3.3 Procedural Object-Oriented Code

Whereas the object-oriented paradigm advocates well-designed, modular software programs, object-oriented programs often demonstrate the absence of object-oriented design because of different reasons.

First, the budgetary and time limitations or the lack of knowledge of object-oriented designers result in the absence of object-oriented design. Budgetary limitations push managers to commence software development phase without properly understanding the domain and decomposing the problem domain into independent classes. The lack of the knowledge of object-oriented design leads to a partial application of the object-oriented concepts. The partial application of the object-oriented concepts produces classes that represent services or subsystems, instead of fine-grained domain objects.

Second, sometimes the usage of the state of the art object-oriented languages is understood to be the only key prerequisite for a good object-oriented design. However, modern object-oriented languages only provide a support for the key object-oriented concepts such as inheritance and polymorphism. Their utilization does not replace the need for a good, upfront design.

Third, constant maintenance efforts erode software design because of additional features added to a program. These additional features may not fit in the existing classes and new classes are needed to be created to accommodate the new changes. The new changes are nonetheless introduced without creating the new classes in software systems, and consequently, the program demonstrates strong coupling, and the presence of misplaced functionality and logic in various classes of a program.

We coined the term *Procedural Object-Oriented Code* (POC) for the code that shows the signs of the absence of the overall object-oriented design, nonetheless developed using the state of the art object-oriented languages. Procedural object-

oriented code often faces similar problems as procedural legacy systems [DDN02]. They both contain duplicated code and logic, misplaced logic, and incomplete abstractions. The term POC makes a clear distinction between good object-oriented code and object-oriented code presenting such defects.

As we shall see in this chapter, the design defects in POC are similar to the design defects described for AntiPatterns [BMMM98]: The design defects in POC also result in huge classes, under-developed inheritance hierarchies, and the lack of abstraction for domain entities. Nevertheless, the design defects in POC distinguish themselves from AntiPatterns by their manifestation in code. That is, the novelty of the POC design defects lies in the new code smells that are symptoms of these design defects. These code smells are considered novel because the AntiPatterns descriptions do not provide all possible manifestations of AntiPatterns in code and different code smells can be identified resulting from the same symptoms as described by AntiPatterns.

POC manifests both the existing set of code smells introduced by Fowler *et al.* [FBB+99] and the new set of code smells that we list later in this chapter. The existing code smells, or object-oriented code smells, are described in detail along with their remedies. It is also necessary to report the novel code smells in POC. The definition of the POC code smells will permit the identification of pertinent algorithms to search for these design defects. In addition, it is also necessary to search for solutions that may help removing the POC code smells from programs. Their removal will help improve the overall object-oriented design of the programs [FBB+99].

Procedural object-oriented code consists of *Partially Decomposed* classes. Partially decomposed classes are huge structures that represent subsystems rather than any particular domain entity. These partially decomposed classes in POC result in architectural-level design defects. These design defects include the absence of class hierarchies. Consequently type and subtype relationships are missing for domain entities *i.e.,* domain entity code is not produced in a hierarchical relationship represented by a parent class and its derived subclasses. Thus, the code related to a domain entity cannot be reused through specialization, a leverage obtained with the presence of hierarchical relationships. In addition, certain domain entities are not represented in their precise classes but their code is scattered across the other classes making up a system.

Figure 3.1 provides a first example of the partially decomposed classes in POC with Polymetric views [LD03]: The figure demonstrates shallow class hierarchies. Moreover, the huge classes in the figure provide a hint about the lack of decomposition of the object-oriented design. We describe these design defects in more detail below.

The POC design defects and code smells are not as widespread as the existing design problems. This is suggested by the widespread literature for the description and detection of the existing design defects and code smells [BMMM98, DDN02, FBB+99, MGMD08, Mar04, TM05]. But we believe that it is essential to bring the POC design defects and code smells forth because these severely mar software

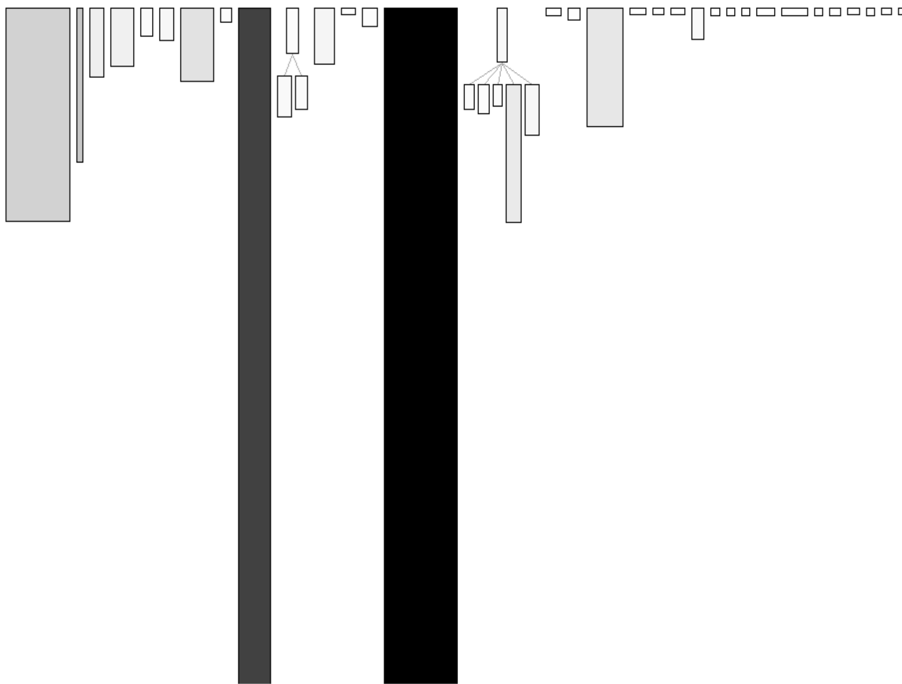

Figure 3.1: Procedural Object-Oriented Code — Rectangles represent classes, edges represent the inheritance relationships between classes, height and width of rectangles is dependent upon number of methods and number of attributes respectively, and node color is determined by the number of lines of code.

design. Consequently, software maintenance efforts prolong for years, as our experience suggests. Hence, it is necessary to report them in the literature even if they exist in a few systems only. Our industrial experience has helped us bringing forth these defects so that others can also benefit from their description, their detection techniques, and the proposals for their correction (cf. Chapter 4).

We detect these code smells while reading code to perform maintenance activities in industrial systems.

In the following sections, we describe in detail various design defects appearing in POC and the consequent code smells.

3.4 POC Design Defects and Code Smells

In this section, the design defects and the code smells occurring in POC are described. But before providing a classification of the POC design defects and code smells, we clarify the precise meanings of the terms design defects and code smells. Design defects are higher-level, architectural flaws in programs, which are also known as, AntiPatterns [BMMM98]. Code smells are the code patterns that arise from the presence of higher-level design defects. This definition of code smells

is consistent with the existing works that propose to formalize design problems in programs [MGMD08, TM05]. These works relate higher-level design defects with object-oriented code smells for the specification and identification of design problems.

Hence, in this section, we present the higher-level design defects in POC and their code-level manifestations.

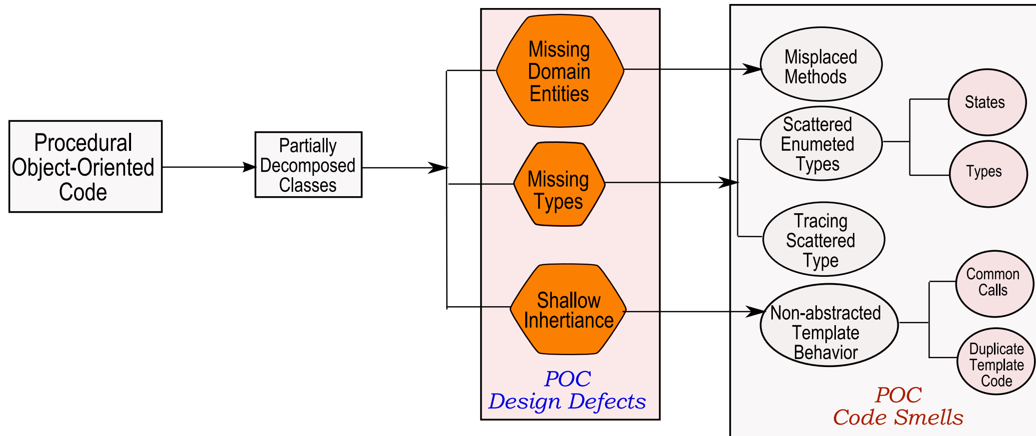

Figure 3.2: Taxonomy of the POC Design Defects and Code Smells

Figure 3.2 demonstrates a taxonomy of the design problems appearing due to the partially decomposed classes in POC. For the taxonomy of the POC design defects and code smells, hexagons represent design defects and ovals represent the code smells that are produced from these design defects. These shapes for design defects taxonomy are first suggested and used by Moha *et al*. [MGMD08] to define a taxonomy for AntiPatterns and code smells.

The taxonomy in the figure describes the presence of Partially Decomposed classes in POC because of the incomplete application of the object-oriented concepts. These Partially decomposed classes lead to the absence of classes for domain entities. Moreover, object-oriented types and subtypes relationships are also absent. The absence of classes for domain entities and types and subtypes relationships for various domain entities leads to the inheritance hierarchies that are not fully developed.

In the sections below, we mention individual design defects appearing in POC in detail and their associated code smells. We also present a comparison of each of the POC design defects and code smells to the existing set of AntiPatterns and object-oriented code smells.

3.4.1 Missing Domain Entities

In POC, there is an absence of the classes for domain entities from the code. The methods associated to these domain entities are scattered across the other POC classes. The Blob AntiPattern [BMMM98] defines a controller class, which contains the logic for surrounding data classes. In POC, there are multiple controller classes and each controller class encapsulates the logic for multiple data classes. This scenario is depicted in Figure 3.3: Classes are represented with rectangles and arcs represent relationship amongst class. Different colors in the controller classes represent the presence of methods related to different domain entities in a single class. The configuration of classes in POC can be considered as a Multiple Blob. The multiple controller classes are aggregated classes: Classes that encapsulate the logic for the other domain entities, as in shown Figure 3.3. Following are the code smells arising due to the missing domain entities.

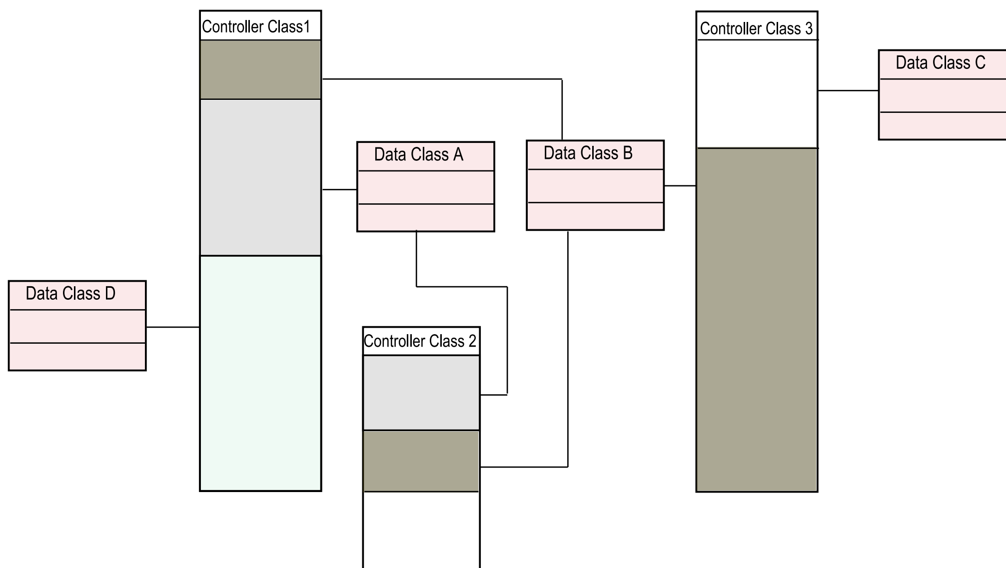

Figure 3.3: Arrangement of Data and Controller Classes in POC

Misplaced Methods

The absence of the class for a domain entity results in the data of the domain entity residing in one data class and the associated operations scattered across the other controller classes. This scenario is depicted in Figure 3.3: The methods operating upon the data in Data Class B are scattered across the three controller classes (depicted by dark grey color). Likewise, for the other data classes, their methods are misplaced. This creates inadvertent coupling amongst many classes

of the system: any change in the domain entity represented by the data class or to its operations cause changes to different classes.

Duplicate Methods

Some of the methods related to domain entities (Data classes) are duplicated in multiple controller classes. This duplication occurs because while looking to perform a particular operation related to a domain entity, developers do not have an idea of the currently implemented operations. In case they are unable to locate the required method, they implement a similar method supporting the desired operation. Hence, some of the operations related to domain entities get replicated in the multiple controller classes.

AntiPatterns and Code Smell Comparison

This design defect can be thought of having some similarity with Blob AntiPattern [BMMM98] and the God class (behavioral form) presented in [Rie96]. However, in POC, there is not a single Blob class, but all the partially decomposed classes make up multiple Blobs. The multiple Blobs complicate the problem as the methods pertaining to the missing domain entities are spread arbitrarily. Therefore, the methods related to the missing domain entities are not found in a single class but these are scattered across multiply Blobs of POC. Hence, while searching to correct this problem by encapsulating common data and behavior in a single class, it would be a mistake to consider a single Blob class. But it will be essential to perform a structural analysis of all the multiple Blob classes to completely move the methods related to a Data class (domain entity) into their appropriate abstraction. A code smell is introduced in [FBB+99] as Data Class, which proposes to move all the relevant methods inside the Data Class.

3.4.2 Shallow Inheritance Hierarchies

Another POC design defect is the absence or scarcity of class hierarchies in software systems. This naturally origins from the omission of the design phase when the common behavior related to subclasses is abstracted in their parent class. Missing parent classes in POC result in non-abstracted template behavior (common behavior) to appear duplicated in the methods of POC classes. Following are the code smells that occur due to the shallowness of class hierarchies.

Common Calls

Template Method [GHJV95] is a design pattern based on the definition of a primitive algorithm in an abstract superclass for a particular process. The skeleton algorithm or the common behavior to be abstracted is described in the parent class. This algorithm is then concretely defined by the subclasses to refine the algorithm according to their own specificity. Template Method is a fundamental technique

for the code reuse. However, the failure to define the skeleton algorithm related to Template Method in a parent class leads to the duplication of "template code" in the methods of POC classes. In such situation, the template code manifests itself in the form of common calls.

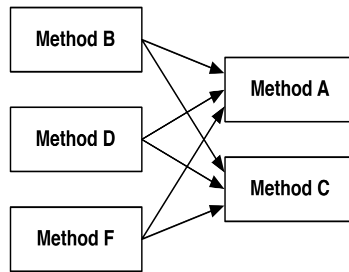

Figure 3.4: Cloned Calls - Missing Template Behavior

Common calls are reflected as duplicate method calls in methods that perform common processes. An example of the common calls code smells is depicted in Figure 3.4. In this case, methods B, D, and F invoke methods A and C. These cloned calls show the presence of non-abstracted logic related to Template Method that can be refactored into a common superclass. Hence, such pattern describes the need to create a common superclass for the methods implementing common calls. The client methods may be present in the same class or different classes.

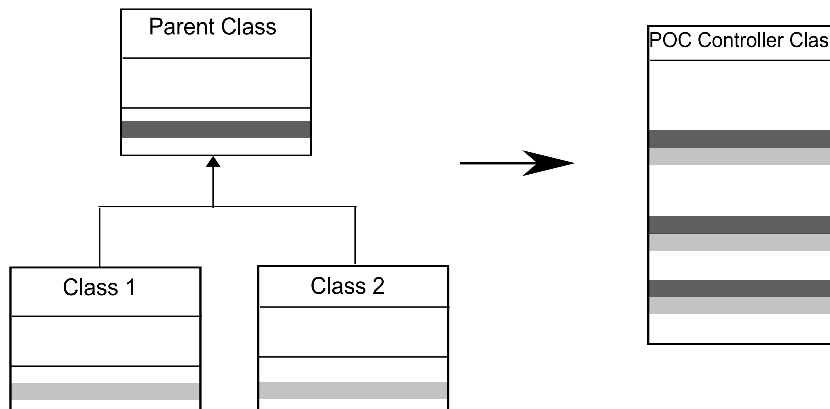

Figure 3.5: Duplicate Template Code

Duplicate Template Code

Duplicate Template Code refers to different methods that implement some common functionalities in addition to their respective specialization. This appears because of the absence of a parent class that could abstract the common logic represented by the duplicate code. In addition, due to the partially decomposed classes in POC, duplicate template logic may be present in a single class. The left side of Figure 3.5 demonstrates a scenario in object-oriented design where a parent class defines a particular behavior (dark gray color). This behavior is inherited and some specialization code is added by each subclass (light gray color). However, the right hand side depicts that in the absence of a parent class and respective children classes, the common code is duplicated in a single POC controller class.

This form is different from the common calls because the duplicate template code surfaces in the form of a set of common instructions working on input variables. These methods are commonly implemented using similar names. This may be overlooked due to the resemblance to method overloading.

We demonstrate this code smell with a code snippet taken from our case study software to create the results of the blood analysis tests. The two different functions from the same class are actually used to create two types of results associated to the tests performed on the patient plasma. The only difference between the two lies in the fact that they create two different objects (line# 16 in Listing 3.1 and Listing 3.2), and the second method sets the error tolerance value for interpreted results (line# 17 in Listing 3.2).

```
1  public resultat CreerResultat(double acceptmin, double acceptmax,double
       ecartmax, double factcor, object redillimbasse,     object
       redillimhaute, object txredilbasse, object txredilhaute, int premierpt
       , int secondpt, test t)
2  {
3    resultat res = mDM.Newresultat(t);
4    res.acceptationmin = acceptmin;
5    res.acceptationmax = acceptmax;
6    res.ecartmoymax = ecartmax;
7    res.facteurcorrection = factcor;
8    if(redillimbasse != DBNull.Value)
9      res.limiteredilutionbasse = Convert.ToDouble(redillimbasse);
10   if(redillimhaute != DBNull.Value)
11     res.limiteredilutionhaute = Convert.ToDouble(redillimhaute);
12   if(txredilbasse != DBNull.Value)
13     res.tauxredilutionbasse = Convert.ToDouble(txredilbasse);
14   if(txredilhaute != DBNull.Value)
15     res.tauxredilutionhaute = Convert.ToDouble(txredilhaute);
16   cinetiquedodeuxpts cine2p = mDM.Newcinetiquedodeuxpts(res);
17   cine2p.premierpoint = premierpt;
18   cine2p.secondpoint = secondpt;
19   return res;
20 }
```

Listing 3.1: Method Duplication

```
1  public resultat CreerResultat(double acceptmin, double acceptmax,double
       ecartmax, double factcor, object redillimbasse,     object
```

```
    redillimhaute, object txredilbasse, object txredilhaute, double
    linearitemin, int premierpt, int secondpt,  test t)
2 {
3   resultat res = mDM.Newresultat(t);
4   res.acceptationmin = acceptmin;
5   res.acceptationmax = acceptmax;
6   res.ecartmoymax = ecartmax;
7   res.facteurcorrection = factcor;
8   if(redillimbasse != DBNull.Value)
9     res.limiteredilutionbasse = Convert.ToDouble(redillimbasse);
10  if(redillimhaute != DBNull.Value)
11    res.limiteredilutionhaute = Convert.ToDouble(redillimhaute);
12  if(txredilbasse != DBNull.Value)
13    res.tauxredilutionbasse = Convert.ToDouble(txredilbasse);
14  if(txredilhaute != DBNull.Value)
15    res.tauxredilutionhaute = Convert.ToDouble(txredilhaute);
16  cinetiquedo cinedo = mDM.Newcinetiquedo(res);
17  cinedo.linearitemin = linearitemin;
18  cinedo.premierpoint = premierpt;
19  cinedo.secondpoint = secondpt;
20  return  res;
21 }
```

Listing 3.2: Method Duplication

AntiPatterns and Code Smell Comparison As far as existing works are concerned, no work explicitly mentions the code smells appearing from the absence of class hierarchies. The Spaghetti Code AntiPattern [BMMM98] includes this description: “Benefits of object orientation are lost; inheritance is not used to extend the system; polymorphism is not used”. This does provide an idea about the absence of hierarchy but the consequent code patterns are not mentioned. Duplicate code smell [FBB+99] is reported in a general manner without a mention of its appearance in methods that provide a hint of the presence of non-abstracted template logic. Extract Subclass refactoring mentions the presence of duplicate code in two classes to be abstracted in a superclass. This refactoring discusses the presence of the duplicate code in two sibling classes. However, in POC, classes are huge structure without any particular focus. Hence, the duplicate template code pointing towards the absence of the parent class can be found in a single POC class or multiple, unrelated POC classes.

3.4.3 Missing Types

The under-developed inheritance hierarchies and missing domain entities result in the absence of important type and subtype information from POC. Therefore, different operations pertaining to different types are defined with the use of global enumerated types and conditionals. The absence of types and subtypes results in the following code smells.

Global Enumerated Types and States

In POC, global enumerated types are used to hold the i) states, and ii) types of objects. These global enumerated types are used in a large number of methods with conditionals, hence leading to their scattering throughout the code. Now, since these objects are global and visible to all objects present in a system, their code is used and modified by all the classes in POC, making their maintenance or reuse difficult. We illustrate such enumerated types with examples below.

States and Enumerated Types Enumerated types are used to test states of various operations in progress within programs and vary the operations performed according to the current state of the domain entity. These state tests are performed in the form of conditional statements and since the enumerated types are global variables, this creates strong data coupling between these enumerated types and the classes using them. Thus, their code is scattered across all the client classes. Examples of such code snippets are shown in Listing 3.3, Listing 3.4, and Listing 3.5, taken from our case study software. The code snippets perform various functions according to the state of the blood analysis. Each of them is extracted from a different method. The text related to enumerated types defining state appears in bold characters.

```
1  if (currentAnalysis.state == AnalysisState.ToStart ||
2  currentAnalysis.state == AnalysisState.NotPerformed) {
3    ValidateCalib(currentAnalysis.calib);
4    PerformQC(currentAnalysis.qc);
5  }
```

Listing 3.3: State-related Conditionals

```
1  if(currentAnalysis.state == AnalysisState.InProcess) {
2      results = currentAnalysis.GetResults();
3      DateTime dateTest = currentAnalysis.GetTime();
4      if(dateTest == DateTime.MaxValue)
5          UpdateResults(results);
6  }
```

Listing 3.4: State-related Conditionals

```
1  if(currentAnalysis.state == AnalysisState.MissingProduct) {
2    UpdateProductQuantity(currentAnalysis.product);
3  }
```

Listing 3.5: State-related Conditionals

Types and Enumerated Types Global enumerated types are also used with conditionals to perform different operations according to different subtypes related to a domain entity. Below, a set of code snippets depict the use of global enumerated types to define different behavior for various subtypes of calibration tests. The code listings above demonstrate the usage of enumerated types in three different methods in POC.

```
1  if(calibration.type == Calibration.Precalibrated)){
2      LoadCalibrationData();
3      DisplayData();
4  }
```

Listing 3.6: Type-related Conditionals

```
1  if(calibration.type == Calibration.Raw)){
2      ChangeCalibration(calib.id, results);
3      AddNewCalibraton(calib.id, false);
4  }
```

Listing 3.7: Type-related Conditionals

```
1  if(calibration.type == Calibration.Ratio)){
2      ChangeCalibration(calib.id, results,ratio);
3      AddNewCalibraton(calib.id, false);
4  }
```

Listing 3.8: Type-related Conditionals

There are three things that can be remarked in the code snippets listed above to use the usage of the enumerated types for states and types.

- Data Class (domain entity) has an associated state attribute (*currentAnalysis.state*) in the example above.
- Data Class (domain entity) has an associated type property (*calibration.type*) as shown in the example above.
- These states are compared to the global enumerated types to ascertain the current state of the domain entity to vary the operations performed according to different states.
- These state and type conditionals do not exist in a single method. But these enumerated types are scattered across different methods defining the logic for the domain entity.

Scattered Code related to Type information

In object-oriented software systems, one-to-one representation of significant domain entities is present in the form of application classes. In POC, such a mapping is absent as POC classes do not represent a precise domain entity. The information regarding types and subtypes is encoded in global enumerated types. In such a scenario, if operations on domain entities are intended to be traced, this is achieved in an ad-hoc manner: A type service is defined that provides the string representation for the enumerated types emulating the missing type. Therefore, methods calls are made to this type service by different methods in POC whenever they need to trace information regarding a domain entity type.

For example, one such instance was observed in our case study software whereby a glossary class manages the mapping of domain entities. This class only returns string representation for particular enumerated code passed to the methods of this class. Every domain object using this information references to the glossary class. Hence, glossary-provided type information becomes scattered in the system due to the extensive glossary-related method calls making it a scattered functionality. As we discuss in the detection strategy, the glossary-related method calls in some POC objects make 4% of all the method calls, creating strong coupling and reducing reuse and maintenance capacity of the system.

AntiPatterns and Code Smell Comparison

Normally, enumerated types, when used with conditionals, provide the code smell for applying Transform Conditional to Polymorphism, as mentioned in [DDN02, FBB+99]. An example described in [FBB+99] for the types used with conditionals is reproduced in Listing 3.9 for illustration purposes. The form of the enumerated types in POC (for example, Listing 3.3, Listing 3.4, and Listing 3.5) is definitely different from the one described in Listing 3.9 in three ways. First, in POC, the enumerated types are not attached to the object for which they define various states, but these are defined as global variables. Second, the user-defined types, in our case, are only associated a state variable and various states exist as separate global variables defined by enumerated types. Third, the state operations do not exist in a single method, as shown in Listing 3.9, but in several methods. Hence, in our case, the enumerated types end up scattered across several methods of POC classes. This scattered manifestation makes their manual detection difficult because the reengineer is required to look into all the methods that use these enumerated types for defining program behavior.

```
1   int payAmount() {
2       switch (_type) {
3           case ENGINEER:
4               return _monthlySalary;
5           case SALESMAN:
6               return _monthlySalary + _commission;
7           case MANAGER:
8               return _monthlySalary + _bonus;
9           default:
10              throw new RuntimeException("Incorrect Employee");
11      }
12  }
```

Listing 3.9: Example of Types and Conditionals from [FBB+99]

3.5 Detection of POC Design Defects and Code Smells

We have provided some design defects and code smells related to the absence of hierarchical abstractions and the missing class abstractions for domain entities. The next desirable thing is to detect the presence of the design defects and code smells

in code [Ciu99, LM06, Mar04]. Manual detection of all the code smells may be too laborious for a developer to undertake this task. The code of the aforementioned global enumerated types can be scattered across a large number of methods. Moreover, duplicate template logic can be present in several methods in POC classes. Hence there is a need for techniques and tools that can identify the design defects and the code smells present in POC. The tools will help identifying all the locations where these code smells exist, automatically. For this reason, we decided to look for the tools and techniques that may help discover the POC code smells and design defects in programs.

We believe that the symptoms of the design defects appearing in POC are not new. They are already described in the context of AntiPatterns [BW97]. Hence, there are techniques that exist to identify their presence in code with the help of code quality metrics. Visualization techniques aid in identifying the POC design defects such as huge classes and lack of inheritance hierarchies. In Chapter 2, we have demonstrated that polymetric views can help detect the presence of the huge classes and the scarcity of class hierarchies. Another example is demonstrated in Figure 3.6 that shows the presence of huge classes and scarce inheritance hierarchies present in application classes.

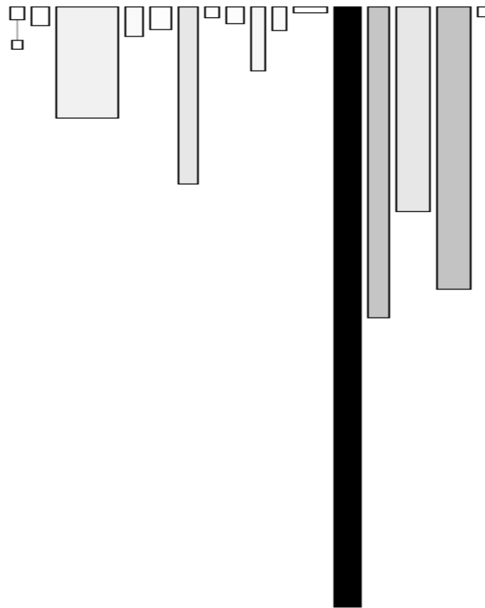

Figure 3.6: Coarse-Grained Polymetric Views - Nodes represent classes, edges represent the inheritance relationships between classes, height and width of rectangles is dependent upon number of methods and number of attributes respectively, and node color is determined by the number of lines of code.

Moreover, duplicate code is a known problem [DDN02, FBB+99] that can occur in various contexts. In this regard, CCFinder is a good and easy to use tool

to search and visualize code clone as dotplots and provide various classes of clones within the same file or in the separate files analyzed [KKI02].

Therefore, some of the above design defects and code smells, namely, huge classes, scarce inheritance links, duplicate template methods, and duplicate methods can be identified with the help of the existing techniques.

Some of the POC code smells exhibit scattered code. Common calls and enumerated types belong to this group. We consider that these code smells cannot be detected with quality metrics as the quality metrics are not apt for the purpose of the detection of scattered code. We need to look into the techniques that search for scattered code to detect these code smells.

In the following section, we describe Scattering Analyzer approach to detect scattered code in POC.

3.5.1 Detecting Scattered Code in POC

In this section, we describe that we need to look for the techniques that are used for detecting scattered code to search for code smells causing code scattering.

Limitations of Metrics and Clone Detection

Code clone detection performs well to detect code clones but we found that developers, in reality, are sufficiently proficient with the refactorings enlisted by Fowler *et al.* [FBB+99] to leave too many duplicate code pieces. The duplicate code is refactored and the duplicate locations are replaced with the method calls to the refactored methods. Therefore, code clone detection techniques may not provide good results where developers have refactored the code clones in helper methods. Hence, a detection technique is needed to be employed to search for these helper methods in application.

The visualization techniques are good at providing depiction of the code quality metrics. However, only those defects can be detected that can be described with code quality metrics. For example, enumerated types replace types and subtypes and these are accessed throughout a program by several methods. These cannot be detected with proposed visualization techniques based on software quality metrics.

Finding Answers with Aspect Mining

We have discussed that the absence of aspects in code is manifested in the form of scattered and tangled code and aspect mining techniques search for the patterns of scattered code (cf. Chapter 2). Hence, aspect mining techniques can equally be utilized to detect scattered code in POC.

One of the techniques to detect scattered code related to aspects proposed the use of identifier aggregates. The purpose is to look for frequently occurring identifiers in code. This idea has been explored by the Aspect Browser tool to look

for the “crosscutting” scattered identifiers in the code [GKY00]; although our experience states that textual code information to ascertain scattering produces too many false positives. For this reasons, scattered types are searched in more recent aspect mining tools looking for identifier scattering [HK01, ZJ04]. We employ the techniques to search for scattered enumerated types.

In addition, the scattered code related to the helper methods can be detected with the Fan-in technique [MDM07]. We consider that this technique can be useful to detect common calls code smell and other types of refactoring of duplicated code.

In the following section, we describe a techniques based on the techniques employed for aspect mining to search the scattered code in POC.

3.5.2 Proposed Approach — Scattering Analyzer

We propose Scattering Analyzer that integrates identifier analysis and Fan-in metric for detecting scattered code in POC. We discuss these two features of the approach in the next subsections.

Identifier Analysis

Identifier analysis proposes a simple algorithm for finding the scattered identifiers in the code: It looks for all the identifiers and searches for those that occur frequently. The developer or maintainer can study these identifiers for problem detection. It provides for an identifier:

- aggregate of the frequency of occurrence of an identifier in the code;
- for each identifier it displays the scattering information, *i.e.,* the class and the associated method which accesses this identifier;

These features help decern the identifiers related to the code smells resulting in scattered code. For example, the global enumerated types are detected using this technique. The scattering information is equally important because one identifier, if encapsulated well, will only be accessed from a single class.

Fan-in Metric

The Fan-in metric calculates total number of calls to a method, to uncover the refactoring of duplicate code in helper methods. We employ Fan-in metric to identify the occurrences of Common Calls and scattered type information code smells in POC by the increased Fan-in metric of the invoked methods.

Figure 3.7 represents our tool based on Scattering Analyzer approach. The tool performs Fan-in analysis and identifier analysis to calculate each identifier and its spread through various classes. We shall describe the tool and its results in detail in Chapter 7.

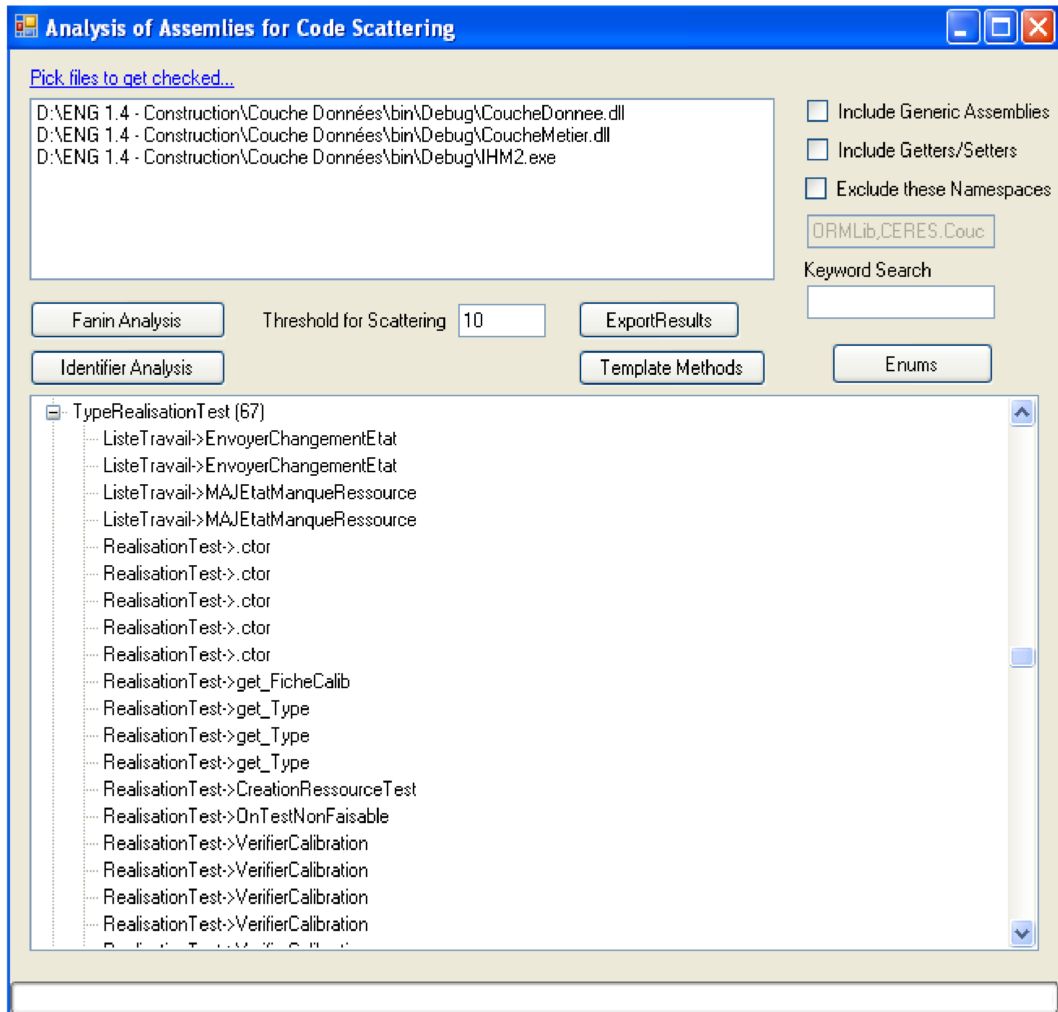

Figure 3.7: Scattering Analyzer

3.5.3 Discussion

Most of the POC design defects and code smells can be detected through the existing techniques or Scattering Analyzer. Still, a concrete limitation of all the detection techniques is the inability to precisely identify the absent domain entities. Such entities require a manual analysis and knowledge of the application domain. Moreover, the techniques discussed should be validated with more case studies.

We believe that the detection of the design defects and code smells related to POC should interpreted as the uncovering of a larger problem: the absence of object-oriented design. All the design defects and code smells are interrelated and the presence of one should indicate that the others are not far behind. Hence, we believe that the system demonstrating POC problems should undergo a major reengineering effort to restructure the whole code. There is a need to resort

to the other, more semantically-rich techniques, such as Formal Concept Analysis [ADN05b, DHHaV04] to find encapsulate data and operations of missing domain entities in new abstractions and finding class hierarchies. Moreover, scattered enumerated types should be encapsulated in their own abstractions to make them compatible with the existing refactorings.

3.6 Conclusion

In this chapter, we have descried procedural object-oriented code, which is the result of the absence of the key object-oriented concepts from software systems. POC consists of Partially Decomposed Classes. POC is shown to demonstrate certain design defects that are similar to AntiPatterns. However, the novelty of the design defects in POC lies in the new set of code smells produced by these defects. These code smells include Duplicated Template Code and Common Calls. The design defects and code smells adversely effect software design. Changes in software systems are scattered, requiring huge effort in terms of time and resources.

Even though, the POC design defects and code smells are not widespread as other design problems, we believe it is essential to bring them forth because these severely mar software design. In turn, software maintenance efforts prolong for years, as our experience suggests. Hence, it is important to report them so that these can be identified and corrective measure be taken.

Various tools and techniques can automate the task of detection of the design defects and code smells appearing in POC. We have shown that existing techniques can be employed to detect the design defects and code smells, which produce similar symptoms as the current design problems. These design defects and code smells can be detected through the usage of software quality metrics and clone detection techniques. However, some of the code smells produce scattered code. For these code smells, we proposed to use the techniques proposed in the domain of aspect mining. Basing itself on the techniques of aspect mining, Scattering Analyzer, permit to search the scattered identifiers and scattered methods calls occurring in POC. This technique helps decern the presence of Scattered Enumerated Types and Common Calls patterns in code.

The technique and tools for the detection of the POC design defects and code smells can be detect individual design defects. However, a more comprehensive strategy is needed to ascertain the overall course for an eventual reengineering activity. In the next chapter, we discuss our strategy to reengineer and restructure POC into an improve object-oriented design.

Chapter 4

Reconsidering Classes in POC

As an understanding of the application improves, the system often needs to be restructured and the abstractions embodied in existing classes often need to be changed. [Opd92]

4.1 Overview

We have described Procedural Object-oriented Code as consisting of *Partially Decomposed Classes*. The hallmark design defects of POC are the presence of huge classes and the absence of class hierarchies. In addition, certain domain entities are not represented in precise classes but scattered in other classes.

We also presented techniques that may be useful to search for the POC design defects and code smells. However, these techniques only detect the POC design defects and code smells. But these techniques are not useful for the correction of these design defects and code smells. POC software requires a major reengineering effort for a better object-oriented design. Hence, it is important to look for technique that can help restructure POC into an improved object-oriented design.

Formal Concept Analysis (FCA) is a mathematical technique to discover useful hierarchical groupings of *objects* having similar *attributes* [GW99]. This technique has been successfully applied to obtain useful groupings of functions and global variables in procedural code to place them in same object-oriented classes [SMLD97, SR99]. This technique has also been applied to analyze and restructure class hierarchies [ADN05a, DDHL96, MHVG08, SS04].

However, the application of FCA for procedural object-oriented code may need further refinements because of the following reasons: First, methods attached to current classes may be misplaced, thus a lattice representing a single class may not provide all the required information. Second, lattices obtained from all the classes in procedural object-oriented code are huge and this vast information may obstruct the analyzer to extract useful groupings amongst the existing classes. In case of reduction of context to find meaningful information, it is necessary to find the pertinent methods within the existing classes that operate upon particular at-

tributes. Third, this type of code may contain interesting traces of object-oriented language constructs such as association and composition relationships, which may be employed to enhance the information to generate class hierarchy. Hence it is not enough to assign procedures to types to get classes as in traditional object identification from procedural code [SMLD97, SR99, vDK99b].

In this chapter, we present a semi-automatic, tool-assisted approach for restructuring object-oriented software showing signs of absence of object-oriented design [BDH08]. We define our approach in four steps:

1. Large classes are decomposed into smaller cohesive pieces. This is achieved by grouping methods present in the code and user-defined types they operate upon in *principal classes* following certain rules.

2. An architectural abstraction for principal classes is obtained to understand the interaction and composition of principal classes amongst themselves.

3. Hierarchies for the methods and attributes of each of the principal classes are obtained by analyzing their accesses to the individual elements of user-defined types.

4. Scattered code related to global enumerated types is identified and refactored into new methods. These methods are then added to the user-specified principal class.

This chapter is organized as follows: Section 4.2 provides a brief introduction of Formal Concept Analysis and Section 4.3 provides a motivation for the definition of a novel approach. Section 4.4 describes our object identification approach for POC. Section 4.5 discusses our approach and its limitations. Section 4.6 concludes the chapter.

4.2 Formal Concept Analysis

Concept Analysis provides a way to identify sensible grouping of *objects* that have common *attributes* [GW99]. A context is a triple $C = (O, A, R)$, where O and A are finite sets (the objects and attributes, respectively), and R is a binary relation between O and A. A concept is a pair of sets: a set of objects (the extent) and a set of attributes (the intent) (X, Y) such that $Y = \sigma(X)$ and $X = \tau(Y)$. Hence, a concept is a maximal collection of objects sharing common attributes. Appendix B provides more detailed information on Formal Concept Analysis.

4.3 Motivation

We intend to create a model based on FCA for procedural object-oriented code. The model should help in the inference of classes and coarse-grained class hierar-

chies from the existing set of POC classes. We lay down the goals for our approach regarding object identification in POC.

4.3.1 Goals of our Intended Model

The model provides following information:

- Composite classes present in procedural object-oriented code can be broken up into more cohesive units.
- Discover scattered data and behavior pertaining to absent classes, and congregate the data and associated behavior in a single class or a hierarchy of classes.
- Finding missing hierarchical abstractions.

4.3.2 Current FCA-based Techniques

Current FCA-based techniques in procedural code do not help attain the above-defined goals. These techniques propose functions as objects and variables as attributes in FCA [SLMM99, vDK99b]. Moreover, proposals for FCA-based class understanding propose methods as objects and class attributes as attributes in FCA. These proposals however have their limitations as we describe below.

In order to demonstrate a lattice formation for a class in POC, we describe our context as follows.

- O = Methods of a POC class
- A = All user-defined types (global variables)
- R = A method accesses or modifies a user-defined type

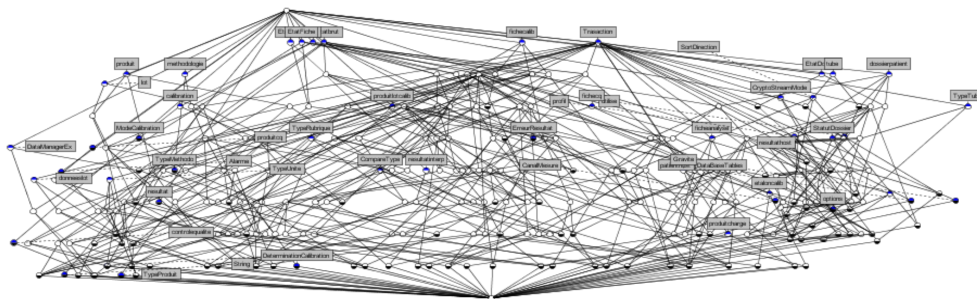

Figure 4.1: Subsystem Classes Access and Modify

Figure 4.1 demonstrates the lattice for one of the classes present in a POC program. As evident from the figure, the lattice presents enormous amount of information. Thus, we identify three main limitations for the application of existing FCA techniques on POC.

- The lattice obtained cannot be used directly to identify any useful information. No interesting grouping of classes and no class hierarchies are possibly extracted from such lattice.
- For context reduction purposes, if only those methods are considered which modify various types, a minority of all the methods is present in the FCA.
- Methods related to a particular type may not reside in a particular class due to the misplaced methods. These types can be accessed and modified from all the methods of all POC classes. Hence, single class lattices may provide only partial information related to data and its associated behavior.

Thus, a mere crude application of FCA to the classes present in procedural object-oriented code will not produce any meaningful results. Therefore, there is a need to reduce the context information in a meaningful manner to obtain reasonable abstractions for encapsulating methods and variables.

Henceforth, we describe an approach for reconsidering and restructuring classes in POC.

4.4 Object Identification in POC

In this section, we describe our object identification process, which supports the discovery of class abstractions in procedural object-oriented code. The overall approach is presented in Figure 4.2. Based on type/class usage we identify *principal classes*. In the second phase, we identify composition relationships between principal classes based on *common creation* pattern, *i.e.,* types that are created together. The last phase consists of finding hierarchy of attributes and methods assigned to the principal classes. We also perform an analysis of enumerated types to associate each of them with their associated principal class.

This section is structured as follow: Section 4.4.1 presents the rules for the identification of principal classes in POC. Section 4.4.2 describes the process of identification of composition and abstractions of similar composite classes. Section 4.4.3 describes the abstraction of hierarchies from the FCA analysis of methods and attributes making up principal classes. Finally, Section 4.4.4 provides the description of the treatment of enumerated types in POC.

4.4.1 Identification of Principal Classes

We have described that classes for domain entities are missing in POC and this results in misplaced methods scattered in POC classes. These misplaced methods work on data classes (cf. Figure 3.3). Hence, the first step in class restructuring in POC consists of the identification of a cohesive set of methods grouped by data classes (type) usage. The identification helps combining the methods and their associated data related to the missing domain entities in a single class . We consider types as the user-defined classes defining a set of atomic (primitive) attributes.

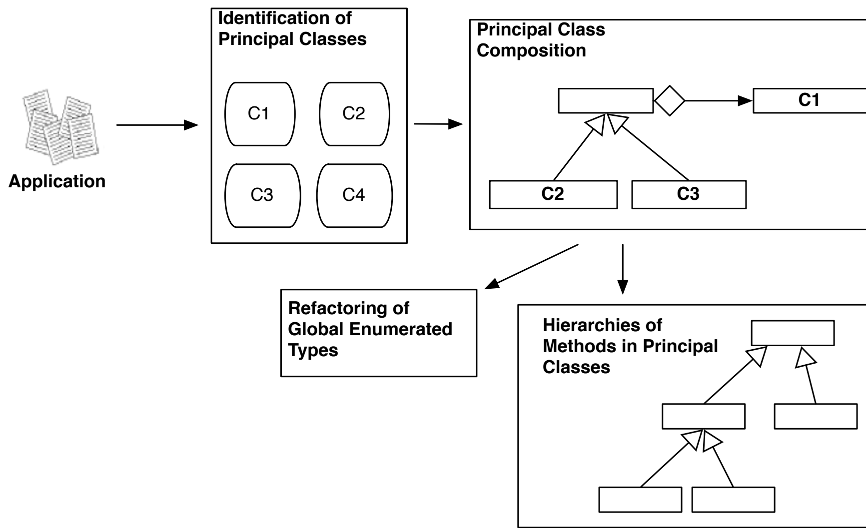

Figure 4.2: Overall Object Identification Approach

Read or write access to an atomic attribute of a type is considered as the read or write access to the type defining the attribute. We term the group of methods and the type that these methods access and modify as a *principal class*. Note that we only consider end-user types as potential target for principal classes, primitive types are not considered, as suggested by Sahraoui *et al.* [SMLD97]. The following rules define method groupings as illustrated in Figure 4.3 using the principle of class cohesion [FP96].

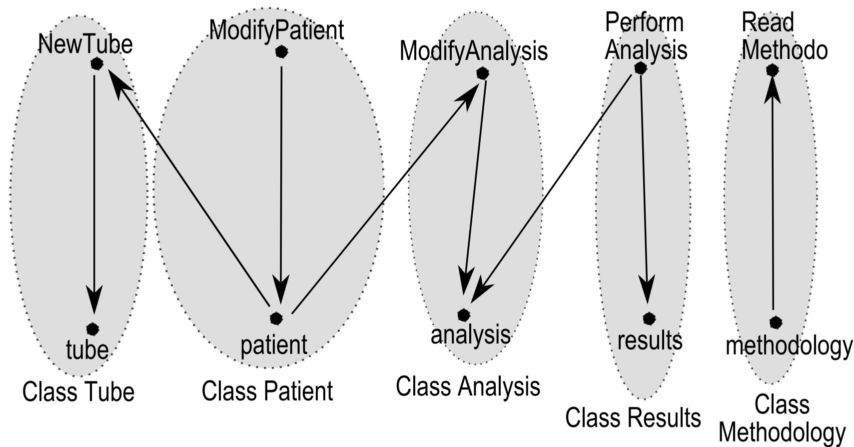

Figure 4.3: Principal Class Identification

- All methods that exclusively write to a particular variable of a given type are associated to the principal class for this type. In Figure 4.3, the method NewTube is associated with the class Tube because it writes to the variable tube of type Tube.

- Similarly all the methods that exclusively read from a variable of a given type are associated to the principal class of that type. The method ReadMethodo is associated to the class Methodology in Figure 4.3 because it reads from the variable methodology of type Methodology.

- A method writing to two user-defined types is considered an anomaly and is not considered in the principal class identification phase. In such a case, the method is marked as a candidate for decomposition using slicing [GL91]. Slicing helps segregate instructions working on different variables. The reengineer is required to refactor code into smaller methods and our approach does not provide a comprehensive strategy for slicing such methods. The method PerformAnalysis in Figure 4.3 is a candidate for slicing. In this case, the method can be decomposed into two different methods and each new method is manually assigned to its corresponding principal class by the reengineer.

- In case a method reads from two types, it is associated to the type with most read number so that the most read type and the method reside in the same class.

- When a method m does not read or write a type but calls another method n in principal class PC1, then the method m is associated to PC1.

From the above rules, we identify five principal classes in Figure 4.3 represented by shaded ovals. This step identifies groups of cohesive entities and methods that make up candidates for principal classes.

During this step, we keep track of the dependencies amongst the identified principal classes. This is achieved by keeping track of the read information or method calls of other principal classes in all the methods of a principal class. For example, the methods of Tube and Analysis principal classes in Figure 4.3 access information of Patient class. This access information is kept and it is later used to identify degree of association amongst principal classes.

4.4.2 Principal Class Compositions

Once principal classes are identified, we identify composition relationships among principal classes so that maximum amount of class interaction information can be integrated in the approach. For this purpose, we identify *Create-Create* pattern in code.

The pattern searches for all the methods belonging to a principal class that create a variable of its own type as well as a variable of another type. For example, a

method belonging to principal class Analysis initializes a variable of its associated type and calls another method which creates instances of another type Results. Thus, a composition link is created such that Analysis is composed of Results as depicted on the left of Figure 4.4. This pattern is also searched in methods marked for slicing because this may provide useful information about principal class composition.

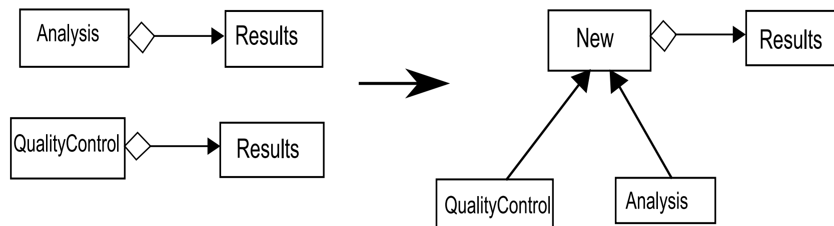

Figure 4.4: Principal Class Compositions

If two principal classes Analysis and QualityControl are found to be composed of a third principal class Results, a new parent class is created which composes itself with Results. Analysis and QualityControl then inherit from the newly created class as demonstrated in Figure 4.4. The parent class helps abstracting attributes in superclasses for optimal class hierarchy. For finding common compositions, concept lattices are created for composition relationship. For this purpose, we define the FCA context as follows:

- O = All Principal classes
- A = All Principal classes
- R = $pc1$ creates $pc2$

Figure 4.5 depicts an example of lattice displaying common compositions (attributes are in grey background, PC represents a principal class). For example, the figure shows that principal classes 5 and 3 commonly create class 2. Moreover, principal classes 1, 3, 4, 7 are not created by any other principal class. So, four superclasses are created containing variables of types of principal classes 2, 8, 6, 5. We shall present some concrete examples of common compositions while presenting the results of our approach in Chapter 7.

4.4.3 Hierarchical Method-Attribute Relationship

Now that principal classes and their composition links are inferred, we identify hierarchical abstractions present *within* a principal class. For this purpose, information regarding the types and methods associated to each principal class is studied.

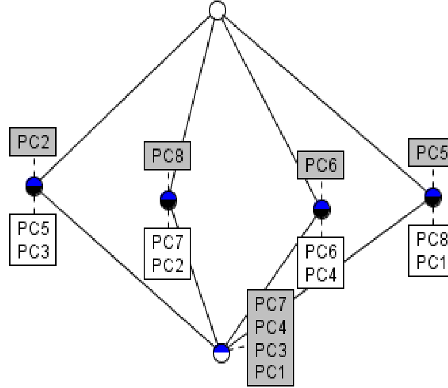

Figure 4.5: Common Compositions

For this, access patterns are observed for methods accessing the attributes of the types in their principal classes. This offers a possible decomposition of principal class into subclasses, with common attributes appearing in the parent class.

We extract using FCA three different views (named fundamental, interaction and associations) to support the possible modularization within principal classes. These views help extract the information depicted in Figure 4.6 from concept lattices to infer class hierarchies within principal classes. Moreover, these views help reducing the structure and information provided by the resulting concept lattices for principal classes. These views can be combined (*i.e.,* a single lattice is generated) for smaller principal classes but for larger ones, extraction of useful modularization in one combined view becomes cumbersome.

Fundamental View. For generating fundamental view lattices, we consider individual (class) attributes of the user-defined types and methods accessing these attributes. When this information is fed into FCA and lattices are generated, the lattices provide hierarchy of methods using the attributes in principal classes. Fundamental view therefore aids in constructing class hierarchy of the principal class so that attribute and method sharing is optimized without cluttering the lattice with principal class access information. For the fundamental view, we define the FCA context as follows:

- O = All Methods within principal class
- A = Attributes of the user-defined type associated to principal class
- R = Method m reads or modifies an attribute of its associated type

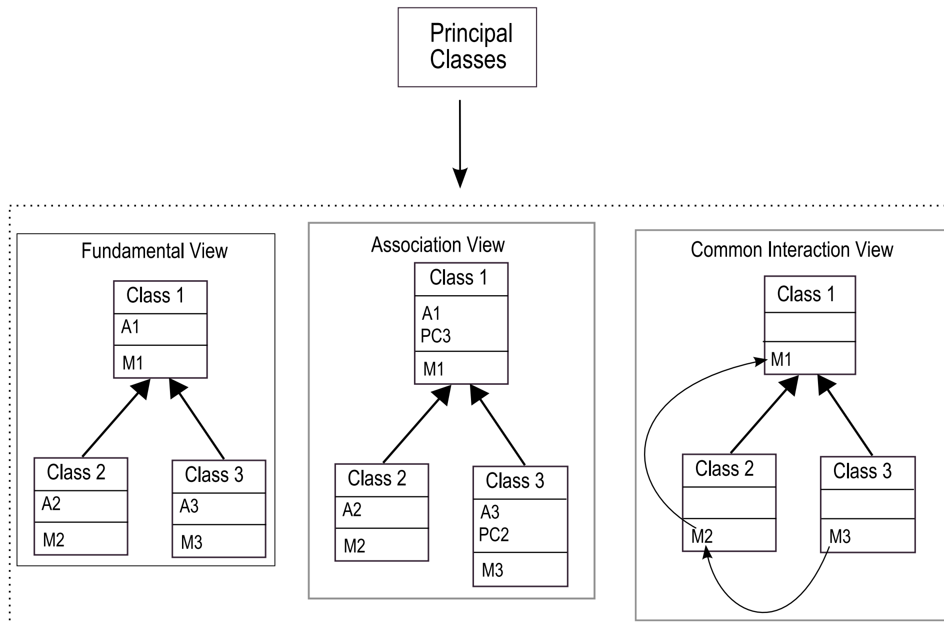

Figure 4.6: Various Views Obtained for Class Hierarchies from Principal Classes

This context has generally been used to search for object-oriented abstractions in procedural code [SMLD97, SR99, vDK99b]. In our approach, the context only contains the attributes and methods related to the principal class in question, hence the results are less complicated.

Figure 4.7 presents an example of a fundamental view of a principal class containing five methods and five attributes (attributes are in grey background). The view provides class hierarchy information for different calibration types and their attribute usage. Calibration type details are omitted purposely because the purpose of the example is to show the inference of class hierarchies from various concept lattice views.

The lattice proposes to constitute the methods and attributes in two disjoints hierarchies: The right-hand side concepts can be restructured as hierarchy of classes with a superclass (2Points:ReadCalib2Points) and a subclass (4Points:ReadCalib4Points), and the left-hand side concepts propose a hierarchy with a superclass (Date:SetDate) and two subclasses (Corrector:CalculateRaw) and (Coefficient:CalculateRatio). This resulting class hierarchy is shown in Figure 4.8. Class Calib is an optional class to bind two different branches of the principal class under analysis. Fundamental views in this case just provide a possible modularization; it is upto the reengineer to decide about Class Calib.

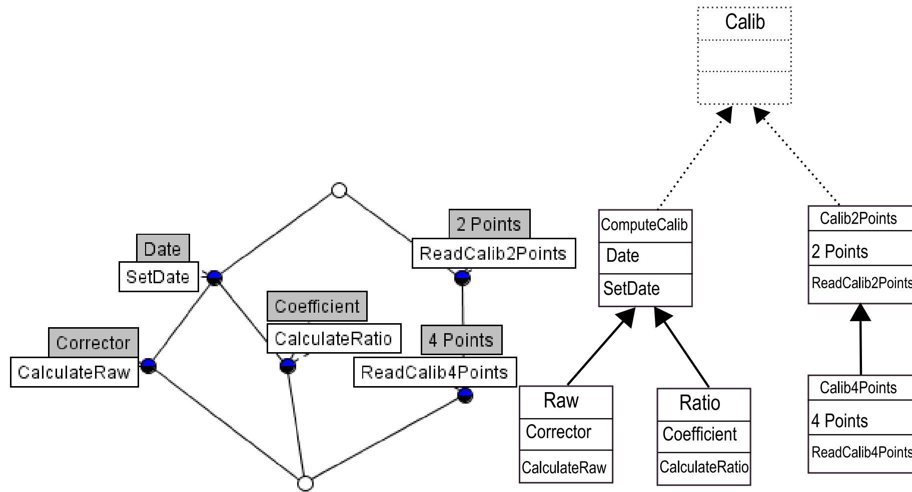

Figure 4.7: Fundamental View

Figure 4.8: Resulting Class Hierarchy

Common Interactions View. The common interaction view helps understanding all of the method invocation of the methods in a principal class. This helps understanding the interaction of various methods present within the principal classes for their possible categorization as interface methods or functionality providers [ADN03]. In addition, this supports the identification of template behavior regarding common method calls, identified in Section 3.4. That is, common interaction view helps identify methods that call common methods: these methods are clustered together.

For this view, we define the FCA context as follows:

- O = All Methods within principal class
- A = Method invocations
- R = method m calls method n

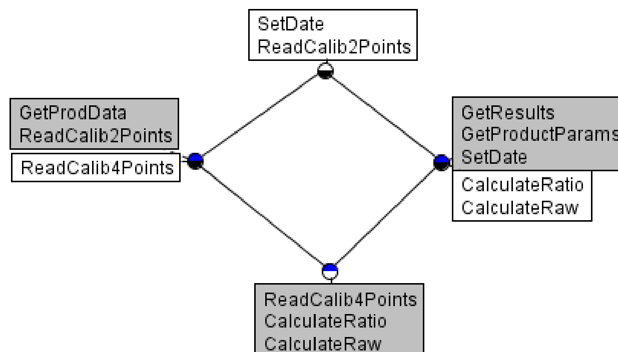

Figure 4.9: Common Calls View of Principal Classes

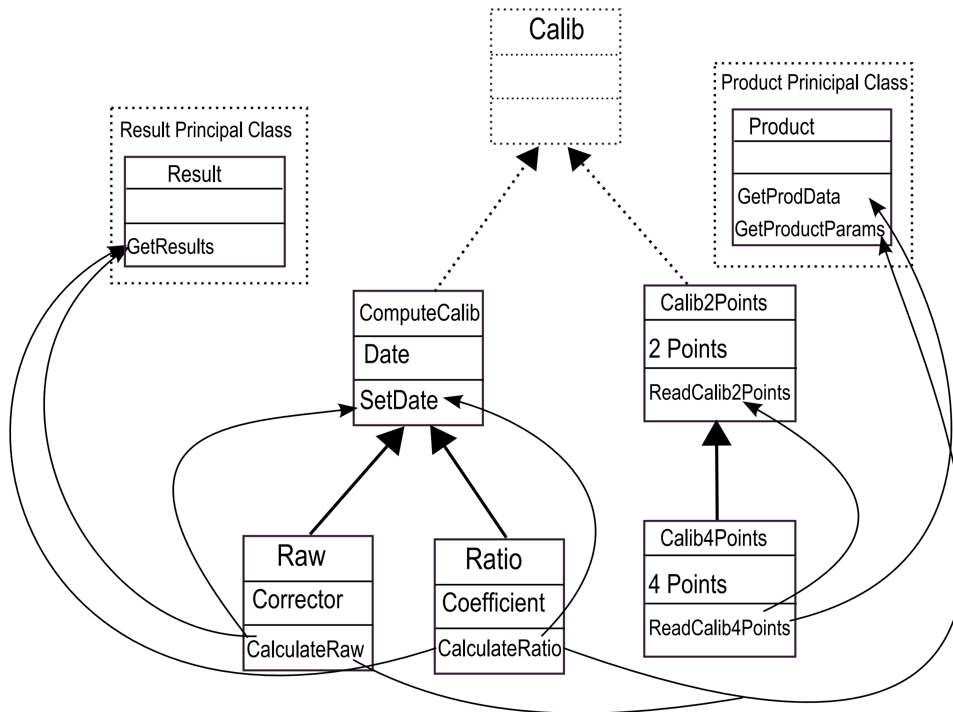

Figure 4.10: Resulting Interactions through Method Calls

Figure 4.9 presents the interaction view concept lattice for the principal class presented in Figure 4.7. The figure demonstrates that method ReadCalib4Points calls ReadCalib2Points and GetProdData. Moreover, the concept containing methods CalculateRatio and CalculateRaw in its extent indicates the existence of common call pattern: they commonly call methods SetDate, GetProductParams, and GetResults. It can also be inferred that methods SetDate and ReadCalib2Points within this principal class do not invoke any methods from inside or outside the principal class. Moreover, method calls to the other principal classes are also indicated in the concept lattice.

The interpretation of the concept lattice vis-à-vis class hierarchy can be understood as shown in Figure 4.10. Method calls present in the concept lattice are graphically shown by pointed arcs for illustration purposes only, our approach does not support this visualization. The methods call to the other principal classes are also demonstrated. In this case, method GetResults belongs to principal class Results while methods GetProdData and GetProductParams belong to principal class Product.

Associations View. In this view, we use the information retrieved regarding relationships of principal classes while assigning methods to principal classes. This information, which we term as *associations*, represents accesses to other principal classes from the methods of the current principal class. For this view, we define the FCA context as follows:

- O = All methods within a current principal class
- A = Attributes of the current principal class, and other principal classes (except those linked by composition)
- R = m accesses an attribute of its type or accesses principal class pc

This view depicts the degree of usage of a principal class within methods of the current principal and hence its place within the hierarchy of the principal class.

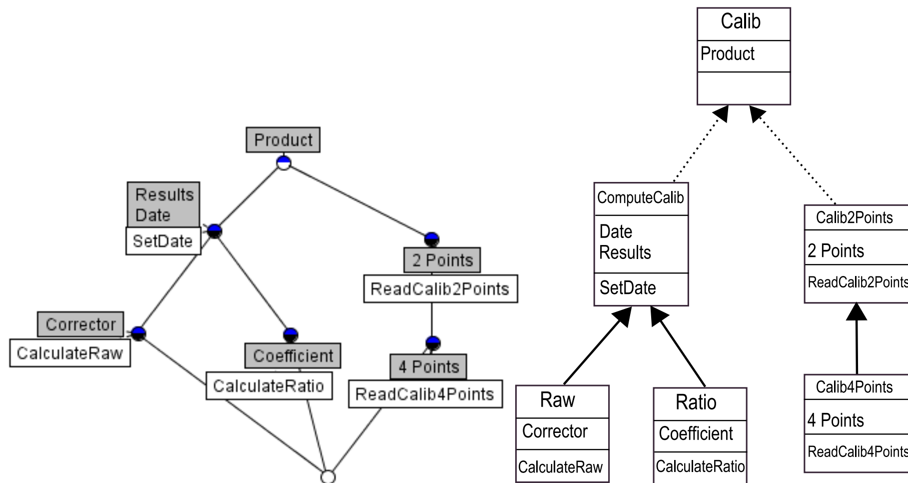

Figure 4.11: Association View

Figure 4.12: Resulting Class Hierarchy

We illustrate this by an example: Consider the lattice presented in Figure 4.11. This lattice augments the fundamental view presented in Figure 4.7. It demonstrates the usage pattern of two principal classes Product and Results within the methods of the current principal class. It is interesting to see that the principal class Product is commonly used in all the methods of the current principal class, while Results is used by another subset. The reengineer can interpret this usage pattern according to her understanding. A new class may be created to include Product from which the two disjoint classes created for the current principal class can be inherited.

The class hierarchy inferred from the association view concept lattice is illustrated in Figure 4.12. The class that was deemed optional in Figure 4.8 is shown to be beneficial because it allows the two hierarchies of the principal class to reuse

the attribute related to the principal class Product defined by the parent class. This improves code reuse in class hierarchies.

In addition, this view helps to segregate the functionality implemented by methods of a principal class. For example, a principal class may have two sets of methods: methods solely working on its attributes and methods which interact with other principal classes. In some cases, these two may represent new "candidate" classes.

Principal classes for which the current principal class already has a composition link are excluded from this view. For example, the current principal class PC1 has a composition link to PC2, then association links to PC2 are excluded from the association view lattice. This is because of the reason that composition relationship is more stronger than association. Composition relations by default appear in top most classes of principal classes.

4.4.4 The case of Enumerated Types

As we defined in Section 3.4.3, global enumerated types with conditional appear in POC to replace the absence of types and states related to various domain entities. We propose a three-step approach to rectify the problem of global enumerated types and states scattered in the code. Our purpose is to change the enumerated types and related code in such a way that refactorings proposed in [DDN02, FBB+99] can be applied. The example presented in Section 3.4.3 is reproduced below. This is the example that we shall use to illustrate our approach. The example is related to the usage of states with conditional to define operations for various states of a domain entity.

```
1 if (currentAnalysis.state == AnalysisState.ToStart ||
2 currentAnalysis.state == AnalysisState.NotPerformed) {
3   ValidateCalib(currentAnalysis.calib);
4   PerformQC(currentAnalysis.qc);
5 }
```

Listing 4.1: State-related Conditionals

```
1 if(currentAnalysis.state == AnalysisState.InProcess) {
2     results = currentAnalysis.GetResults();
3     DateTime dateTest = currentAnalysis.GetTime();
4     if(dateTest == DateTime.MaxValue)
5         UpdateResults(results);
6 }
```

Listing 4.2: State-related Conditionals

```
1 if(currentAnalysis.state == AnalysisState.MissingProduct) {
2   UpdateProductQuantity(currentAnalysis.product);
3 }
```

Listing 4.3: State-related Conditionals

The following approach can equally be applied to scattered enumerated types relate to states and types. Here our examples only illustrates their usage with scattered conditionals related to states. We define these steps below.

Search. Enumerated types are used as global variables in different methods and the logic associated to a particular enumerated type is scattered in these methods. Hence, the first step is to decern the location of global enumerated types and refactor these enumerated types in methods. For this purpose, a simple tool is developed that inspects usage of enumerated types along with conditional statements for guiding the identification of such patterns to ease location identification of places where these enumerated types are used. We find the code snippets in Listing 4.1, Listing 4.2, and Listing 4.3 with the help of the tool.

Merge. The second step is to merge the domain entity defining the state (resp. type) variable and various states (resp. types) defined in enumerated types. This merge is achieved by moving the enumerated type defining state (resp. type) for the domain entity into the class defining the domain entity. We elaborate this step with our examples described in Section 3.4.3 and shown in Listing 4.1,Listing 4.2, and Listing 4.3. For states enumerated type, it is associated to the *analysis* class, hence its definition is moved inside the definition of the *analysis* class. The refactored code is then only visible to the class defining it. The refactored code is shown in the example below.

```
1  public class Analysis {
2   enum AnalysisState {
3       ToStart ,
4       NotPerformed ,
5       MissingProduct ,
6       InProcess
7       };
8  }
```

Encapsulate. Once the global enumerated type is moved to its particular class, the third step consists of encapsulating the test conditions that are related to the states (resp. types) in a single method. This encapsulation will help refactoring scattered conditional in a single method. Moreover, it brings the problem to the one defined in [FBB+99] and the associated Replace Type Code with State/Strategy can be applied. This is shown below for the aforementioned example

```
1  public int ScheduleAnalysis() {
2
3  if (currentAnalysis.state == AnalysisState.ToStart ||
4  currentAnalysis.state == AnalysisState.NotPerformed) {
5    ValidateCalib(currentAnalysis.calib);
```

```
6     PerformQC(currentAnalysis.qc);
7   }
8   if(currentAnalysis.state == AnalysisState.InProcess) {
9       results = currentAnalysis.GetResults();
10      DateTime dateTest = currentAnalysis.GetTime();
11      if(dateTest == DateTime.MaxValue)
12          UpdateResults(results);
13  }
14  if(currentAnalysis.state == AnalysisState.MissingProduct) {
15    UpdateProductQuantity(currentAnalysis.product);
16  }
17
18  }
```

The methods that are created for encapsulating logic for various states of domain entity are then associated to the principal class identified for the domain entity. Places from where these enumerated types are extracted, these are replaced with method calls to the newly created method.

4.5 Discussion

The approach helps identifying objects missing in procedural object-oriented code. User-defined types and methods are used to extract class hierarchies from POC. This is done by analyzing the usage of atomic attributes in a principal class by the methods of that principal class. Moreover, method invocations and the degree of principal classes association help detect common interactions and identification of association degree of various principal classes.

The approach should be applied first to identify principal classes in code and their composition relationships. These two provide information regarding classes and their interactions. Once this information is obtained, each principal class should be individually examined for the class hierarchy of its attributes and methods. For the extraction of class hierarchies, the three views are used to construct classes and subclasses within the principal class. Finally, global enumerated types are considered to augment the class hierarchy information obtained from principal classes and FCA lattices.

Various views are generated to understand the internals of principal classes. We believe that these views do not automate the task of object identification. They act as a tool to guide the restructuring activity. Reengineer needs to interpret different views to combine them in an intelligent manner as the three views are not completely orthogonal. First, she needs to understand the compositions. Once compositions are inferred, hierarchies of methods and attributes in principal classes are identified with the fundamental view. Later, new classes obtained from common interaction view and associations view should be integrated to the overall class model of the principal class in question. Common interaction view should also be considered to understand the collaborations amongst methods of a principal class.

It is not a trivial task to restructure POC classes. However, no approach supports the restructuring of these partially decomposed classes into a meaningful class hierarchy. The object identification approach presented in this chapter actually complements earlier work on refactoring [FBB+99] by providing a framework of class hierarchies in which to perform those refactorings.

The identification of principal classes and composition relationships are automated steps of the approach. Hence, they require minimum human involvement to extract meaningful results. A reengineer is only required to validate the results and to remove anomalies such as methods writing to variables of two different types.

We believe that the main limitation of the proposed technique is that the reengineer should interpret concept lattices to obtain useful information from them: The third step of the approach requires human intelligence to extract useful grouping and hierarchies of classes. Heuristics and guidelines can be developed for the extraction and refinement to identify objects from fundamental, association, and common interaction views as described in [SLMM99]. However, such heuristics are application dependent. We provide general views from which the reengineer can infer class hierarchy information and refine it with domain knowledge.

The variables (of user-defined types) are of both declared and instantiated types. The important point is that a method is considered as reading or writing a particular type whenever the method reads or writes the type by using its reference or its value in memory. We consider these implementation details and they are omitted to make the model more general. Method calls to getter and setter are only considered when there is no direct access to the instantiated variable of a user-defined type. This is covered by the last rule for method assignment to principal classes.

The presence of user-defined types, groupings of primitive types for the representation of domain entities, is essential for the approach to identify principal classes. In addition methods are assumed to be crisp in their functionality in that they implement functionality pertaining to a particular task and they are not huge. In the absence of these two elements, the restructuring approach would fail because of failure to identify principal classes and failure to correctly assign methods to principal classes. If methods are huge providing complex functionality, slicing [GL91] and refactoring [FBB+99] should be applied first to split chunks of related instructions into methods.

4.6 Conclusion

Procedural object-oriented code appears due to absence of software design or due to its erosion over a period of time. This chapter describes our approach for the identification of useful abstractions in procedural object-oriented code. For this purpose, principal classes are identified, their composition links are discovered and the hierarchical relationship of their methods is identified through the usage of FCA. Various views are provided with different information to infer hierarchies

of methods and attributes. Concept lattices provide us with several modularization proposals for methods and attributes present in principal classes. Our approach bases itself on the presence of user-defined types in procedural object-oriented code. The benefits of using concept analysis is to reduce the analysis of huge amount of lines of code to a few high-level design choices. In their absence, a manual task is required to identify them. Our approach also assumes presence of well-focused methods.

Chapter 5

Scattered Concerns in POC

Any given problem involves different kinds of concerns, which should be identified and separated to cope with complexity and to achieve the required engineering quality factors, such as adaptability, maintainability, extendibility and reusability.
[Dij76]

5.1 Overview

A concern is defined as "any matter of interest in a software system" [FECA05]. Modularizing different concerns leads to the improvement of software quality and provides many benefits, including reduced complexity, improved reusability, and simpler evolution [KLM+97]. Aspect-Oriented Programming (AOP) origins from the fact that there are some concerns that remain scattered in the presence of good design because existing programming techniques are insufficient to capture them in useful abstractions [Kic96]. The inventors of AOP stated that without the application of these appropriate abstractions for the intricate concerns, these concerns manifest themselves as scattered and tangled code: Thus non-AOP applications manifest "crosscutting" concerns: a concern whose implementation spans multiple program entities [FECA05]. Aspect mining refers to the search of scattered and tangled code originating from these crosscutting concerns so that these can be appropriately abstracted with aspect-oriented programming concepts. However, aspect mining techniques have explored the hypothesis that the scattered code only originates from the absence of appropriate AOP abstractions and the application does not lack OO design.

In this chapter, we examine the problem of scattered code in procedural object-oriented code through the prism of aspect mining techniques and from the crosscutting concerns point of view. We describe the behavior of aspect mining techniques when object-oriented software systems suffer from the lack or erosion of object-oriented design. The purpose is to understand various crosscutting candidates identified by aspect mining techniques in POC. The contribution of this chapter is two-fold. First, we demonstrate the limitations of current aspect mining

techniques in identifying aspect candidates in POC. Second, we examine various causes of scattered concerns in POC and provide a taxonomy for aspect mining results obtained [BD08].

5.2 Aspect Mining in Procedural Object-Oriented Code

POC consists of scattered code appearing due to the absence of object-oriented design because all domain entities do not have their corresponding abstractions and class hierarchies are missing. Moreover, absence of aspects also results in the appearance of scattered code related to crosscutting concerns. Hence, we mine scattered code in POC using aspect mining techniques for two reasons. The purpose of the study is two-fold:

1. The first purpose is to understand the origins for scattered code appearing in POC through the study of the mined crosscutting concerns seeds.
2. The second purpose is to provide an elaborated taxonomy of crosscutting seeds identified with aspect mining techniques in POC.

While searching for the techniques to mine aspects in POC, we looked for the techniques that have tool support readily applicable to our case study software because C# language is not supported by many aspect mining tools [KMT07]. We found that Aspect Browser and clone detection techniques have tool support that is language-agnostic and these were directly applicable to our software system in question. However, clone detection techniques do not provide many aspect candidates where cloned code is already refactored into helper methods. Therefore, we decided to explore the Fan-in technique because the Fan-in technique identifies aspect candidates that are refactored into helper methods [MDM07]. In addition, this technique can mine meaningful aspects, directly in code, without the need to execute software system and the Fan-in technique seems to common to various aspect mining techniques [GK05, MDM07, BZ06].

Hence, we used two aspects mining techniques to mine scattered concerns in procedural object-oriented code. One tool was selected from each of the categories of aspect mining techniques: dedicated code browsers and automated techniques, represented by Aspect Browser and Fan-in metric, respectively.

One vocabulary point: In this chapter, we use the term crosscutting concerns for all the candidate concerns identified by aspect mining techniques, whether they appear due to the absence of objects or aspects.

In the following, we present a description of the techniques used for mining aspects and the results that we obtained using each technique.

5.2.1 Aspect Browser

Aspect Browser uses lexical pattern matching for querying the code, and a map metaphor for visualizing the results [GKY00]. It extracts fragments of identifier

names from source code according to a programmer specified naming convention. The results are reported as a list of aspect candidates.

Aspect Browser calculates two metrics for the source code in question:

- redundant lines;
- most common identifiers.

Aspect Browser extracts information from source code of a program, and therefore comments and keywords are also considered while calculating the metrics. Hence, the two metrics sometimes carry useless information related to comments and the keywords like "new" and "return". A manual effort is required to filter and aggregate information related to the identified concerns. Aspect Browser (like other dedicated code browsers) is dependant on the naming convention used in the source code, and assumes that implemented crosscutting concerns have a similar keyword (signature) in all classes. In the absence of code naming conventions, aspect browser would not reveal identifier related to crosscutting concerns. When an aspect candidate is found in aspect browser, the matching text is highlighted in the source code and the tool will indicate the match-count.

The tool displays the query results in a Seesoft-type view as highlighted strips in enclosed regions representing modules (e.g., compilation units) of the system [ESEE92]. This helps provide a view in which each file is represented as vertical strips, where a row of pixels in the strip represents a line of code. Figure 5.1 illustrates the mining results for two scattered concerns in Aspect Browser. Existence of searched identifiers in files is highlighted by their corresponding colors. Two identifiers searched in this case are *DebuterTransaction* and *resultatbrut* (checked in the left pane in Figure 5.1) pertaining to transaction and raw results concerns respectively. Their coloring indicate the presence of these identifiers in various files. Hence, Aspect Browser can help reckon the scattering of identifiers and tokens related to a concern.

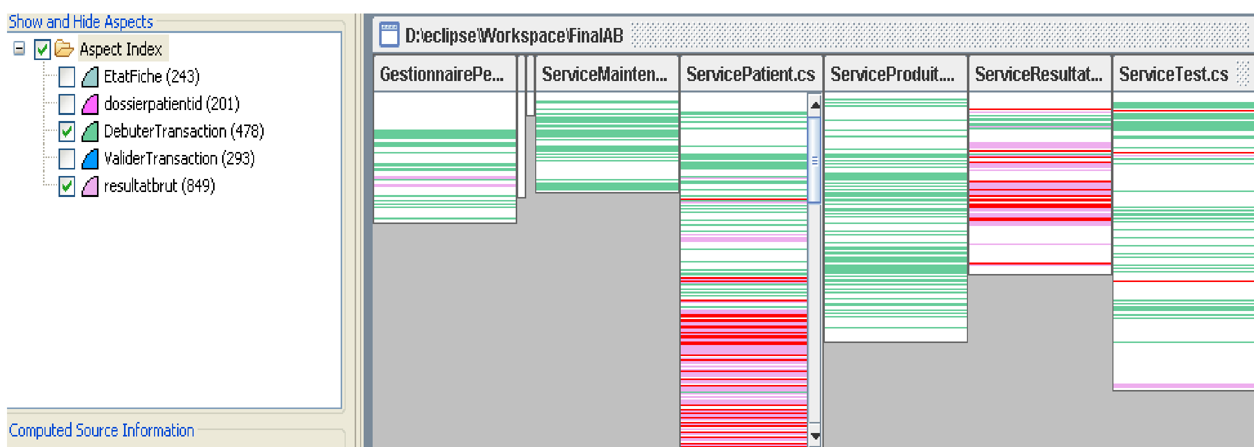

Figure 5.1: Scattered Concerns in Aspect Browser

5.2.2 Aspect Browser Results

Table 5.1 demonstrates the concerns that are mined in our case study software using the tool and the frequency of the occurrence of the identifiers and tokens related to each concern. All the identifiers and lines mined with Aspect Browser are associated to a particular concern by looking at their names and associating them manually with the concerns. The third column of the table indicates a possible classification of each scattered concern: Each crosscutting concern identified by the tool is classified as related to *domain* or *aspect*.

The classification is performed manually by looking at the results of Aspect Browser. The classification pertains to the recognition of the name of domain entities in the list of scattered concerns. For example, Patient, Product, and Analysis are domain concerns of our software system. Hence, a domain expert can perform the classification task to identify the domain entities in the list of crosscutting concerns.

We classify them as domain because these scattered concerns appear due to the absence of their object-oriented abstractions. Hence, the classification specifies the appropriate abstraction for encapsulating each of the scattered concerns list in Table 5.1. Different lines and identifiers referring to the same concern in Aspect

Table 5.1: Crosscutting Concerns and their Frequency

Concern	Frequency	Possible Classification
PhysicalMeasures	37	Domain
Trace	40	Aspect
Events	91	Aspect
Singleton	118	Aspect
Glossary	222	Domain
Quality Control	240	Domain
Analysis	280	Domain
Transaction	300	Aspect
Product	350	Domain
Calibration	550	Domain
Patient	555	Domain
RawResult	750	Domain

Browser are aggregated under the name of the referred concern in the table. The frequency of the concern is calculated by adding up the two metrics presented by the tool for each concern's related lexical tokens and discarding the repetitive entries. The metric represents the total number of artifacts for each concern identified by Aspect Browser. Thus, the metric shows the extent of scattering for each concern. We also manually applied Porter stemming algorithm algorithm [Por80] to group identifiers with similar roots, as performed in [TM04]. For example, identifier 'calib' and 'calibration' are considered belonging to the same stem, calibration.

Below are the results for some of the concerns found in the system.

Overall, 66% of the crosscutting concerns identified by Aspect Browser are related to domain entities of the case study software, as demonstrated by the possible classification of these concerns in Table 5.1. This is expected in POC because domain entities do not have their corresponding abstractions and their code is scattered in other classes. This can be an interesting result from object-oriented refactoring point of view to demonstrate the need to modularize scattered code in objects. However, while searching for aspects, this adds up too much burden on aspect miner to filter these results. In some cases, it may lead to wrongly "aspectize" code related to absent domain entities.

Hence, aspect mining results in POC constitute a huge amount of false-positives related to domain entities in the aspect candidates. These false-positives surface because Aspect Browser only searches for scattered identifiers and tokens regardless of their origin: absence of objects or aspects.

5.2.3 FAN-in Metric

Fan-in analysis fits in the category of automated aspect mining approaches. The basic assumption for the Fan-in aspect mining technique is that the scattered code has been factored out into helper methods [MDM07]. These methods are invoked from all the places where the scattered functionality is required, hence such methods are called more frequently than the other methods from many places, giving them a high fan-in value.

Fan-in aspect mining technique uses the Fan-in metric. The Fan-in metric is a "measure of the number of methods that call some other method" [Som00]. Fan-in collects the set of (potential) callers for each method and the cardinality of this set gives the required Fan-in value. Fan-in analysis consists of the following:

1. Computation of the fan-in metric for all methods.

2. Filtering of the set of methods to obtain the methods that are most likely to implement crosscutting behavior.

3. Analysis of the remaining methods to determine which of them are part of the implementation of a crosscutting concern.

However, this technique has only been validated upon case studies demonstrating good Object-Oriented design [MDM07]. We applied the Fan-in aspect mining technique to POC. Figure 5.2 demonstrates interclass and intraclass interactions between classes through method calls in POC (a snapshot taken using MOOSE reverse engineering environment [MGL06]). The big rectangles in the figure represent classes, the small ones represent methods in the classes, and the lines amongst each small rectangle depict method calls.

Now, we would like to identify and analyze scattered method call candidates identified by Fan-in in such code. For this purpose, we developed a tool to support

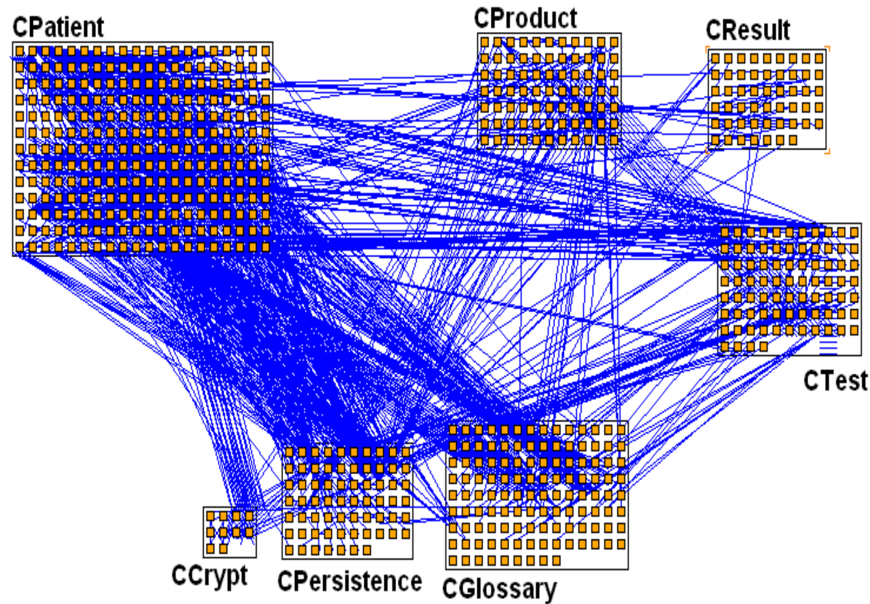

Figure 5.2: Inter-Class and Intra-Class Method Invocations in POC

the Fan-in technique for C# language because there is no tool for computing FAN-in in C#. The tool carries out all the three tasks defined above for the identification of scattered method calls.

5.2.4 FAN-in Results

The tool looks for method calls to all the methods defined in the application classes and lists those with values higher than the filtering threshold. The filtering threshold value is given by aspect miner for the degree of their scattering *i.e.,* Fan-in metric value. In our case, we set 10 as the filtering threshold for mining aspect mining candidates, as suggested in [MDM07]. Table 5.2 shows the crosscutting candidate methods (pertaining to crosscutting concerns) identified by our tool and their Fan-in values along with their possible classification.

The table includes various methods, we describe a few to illustrate their purpose in the code and illustrate the reasons for the increased Fan-in metric.

A first example is that of methods that search or read data. There are five methods that search data related to domain entities. For example, SearchPatient method looks for patient data and SearchProduct looks for product data. Now, these methods are clearly related to the domain entities of the software system, as they search domain entity data. The calls of these methods are scattered because there is no class that encapsulate the operations of the associated domain entity. For example, the calls of SearchPatient are scattered because there is no class that encapsulates

Table 5.2: Application methods and associated Fan-in values

Method	FAN-in	Possible Classification
UpdatePhysicalMeasures	10	Domain
CreateResultCalibration	10	Domain
NewMeasureCalibration	10	Domain
SearchProductIndex	10	Domain
SearchCalib	13	Domain
SearchPatient	17	Domain
PublishException	19	Aspect
ReadMesureCalib	22	Domain
Trace	24	Aspect
SearchProduct	26	Domain
SearchTestData	29	Domain
DecryptData	35	Aspect
ReadRawResults	41	Domain
PublishEvent	96	Aspect
ValidateTransaction	89	Aspect
GetGlossaryValue	127	Domain
GetInstance	101	Aspect

patient data operations. Hence, SearchPatient operation is called arbitrarily by all the methods whenever they need to look for patient data. Thus, the calls of the method are scattered not because it crosscuts different concerns, but it appears scattered because the operation is not called encapsulated in its own class.

A second example of a method with a high Fan-in metric and related to domain entity is that of CreateResultCalibration. This method is invoked to create results of calibration tests. As there are different types of calibration types present in the software system, and each of them creates calibration results, hence this method is invoked several times. The increased number of method calls appear because of the lack of a parent class for calibration types that could abstract their common behavior, such as creation of results, in a template method.

A third example is that of UpdatePhysicalMeasures method. The method is used to update results of the three different types of tests present in the software system, namely, QualityControl, Analysis, and Calibration. Whenever each of these tests finishes, the measures are updated from the results obtained. The code for the method is shown in Listing 5.1. UpdatePhysicalMeasures method contains logic for the each of the different type of tests. Hence, this method is called by each of the different types of tests and this results in its increased Fan-in metric.

```
1 if (Test.Type == Analysis)
2     UpdateAnalysisMeasures(results);
3 else if (Test.Type == QualityControl)
4     UpdateQualityControlMeasures(results);
5 else if (Test.Type == Calibration)
```

```
6        UpdateCalibrationMeasures(results);
```

Listing 5.1: UpdatePhysicalMeasures Code

We have added a possible classification of the scattered methods according to the domain information. The purpose of adding the classification column in the table is to quantify the presence of domain entity related concerns in the list of all the crosscutting concerns mined with the Fan-in technique.

The classification is based on two criteria. First, we look at the method names and those having similar names as domain entities are marked as potentially related to domain entities. Second, we manually look for the methods and their callers. The methods with high Fan-in numbers that are not encapsulated in their classes or demonstrating code problems such as the one shown in Listing 5.1 are marked as related to domain entities. A list of code problems that we looked for are described later while we describe the taxonomy of the Fan-in results in POC.

Although crosscutting concerns indicated the presence of scattered code, almost 63% of the results pertained to the methods pointing to domain entities because of the non-abstracted domain logic (See Table 5.2).

Hence, it shows that the FAN-in metric can identify different types of crosscutting concerns (pertaining to the absence of aspects and non-abstracted domain entities) but without distinguishing them. This inability to distinguish various scattered method calls is that there is no inherent way while analyzing method calls to ascertain the origin of crosscutting concerns. Too many calls for a method provide us a hint about the possible scattered logic of the method in invoking methods. Aspects may not be the most appropriate solution because the scattered logic in callers may appear because of the non-abstracted domain logic. And it may be better to look for object-oriented refactoring before resorting to aspectized solutions. Thus, Fan-in metric information needs to be complemented with some semantic information to segregated and classify scattered concerns. The usage of the application data (domain entities) may provide useful information because the scattered behavior pertains to domain entities. Hence domain entities can be used to distinguish the type of the behavior being invoked and the type of logic the invoked method provides to its caller.

5.2.5 Comparison of Results

The first and the foremost thing to be remarked in the results of the two aspect mining tools used on our case study software is the presence of domain entities in the list of crosscutting candidates. These false-positive aspects appear because there is lack of elementary design, and scattered artifacts (identifiers and methods calls) are constituents of various domain entities. Scattered identifiers identified by Aspect Browser related to missing objects and aspects are much higher in frequency than those identified by the Fan-in tool. Aspect Browser also identified those methods that were identified scattered by the Fan-in tool. This is because

Fan-in detects crosscutting concerns through methods calls while Aspect Browser searches scattered lexical tokens, which also include scattered method calls.

The results of the Fan-in are more precise. This is because of the reason that Aspect Browser lexical tokens may sometimes contain large number of false-positives due to inclusion of comments, name similarities and the presence of language-constructs. Hence, as suggested in [CMM+05], Aspect Browser (and other dedicated code browsers) could be used to analyze and expand the crosscutting seeds identified by automated techniques beforehand.

5.2.6 Taxonomy of Crosscuttingness in POC

Now that we have seen false-positives related to domain entities in the results of aspect mining tools that we used, we proceed to investigate the various types of "crosscuttingness" or scattered code that is found in the results. A thorough investigation of the crosscutting candidates identified by aspect mining techniques reveals that the application consists of two kinds of scattering: Type scattering and Behavioral scattering. We analyzed each of them and describe them below.

Type Scattering The absence of domain entity abstractions and aspects in POC causes different types of *domain types* and *aspectual types* to appear in a program. This distinction is achieved by looking at various identifiers related to types and distinguishing those related to domain entities.

The scattered domain types are related to the presence of data classes and global enumerated types representing domain entities. Scattered domain types appear in POC because the access and manipulation of the data in data classes is controlled by a set of POC classes. These POC classes encapsulate behavior operating on the data classes. Hence, non encapsulation of the data and behavior in a single class results in the appearance of scattered data classes.

Moreover, global enumerated types replace the absence of types and subtypes, and the absence of states in POC. As stated earlier, POC classes contain a scattered manifestation of the domain types. This separation of data and associated behavior is manifested in Figure 5.3: POC classes encapsulate behavior operating on various types while enumerated types and data classes encapsulate data of the system.

Another scattering is that related to Aspectual types. Aspectual types are those that encapsulate logic related to the crosscutting concerns appearing due to the absence of aspects. These aspectual types have inherent crosscutting in their nature, as suggested in [KLM+97]. These appear because they do not interleave with the primary concerns in non-recurrent manner.

Hence the these scattered types can be divided into three categories.

- **Global enumerated type accesses.** Dispersed accesses to global enumerated types representing the states and object types of various entities (such as patient, test, tube types, etc.) in diverse methods of classes present in the system. Aspect Browser shows them scattered in different classes.

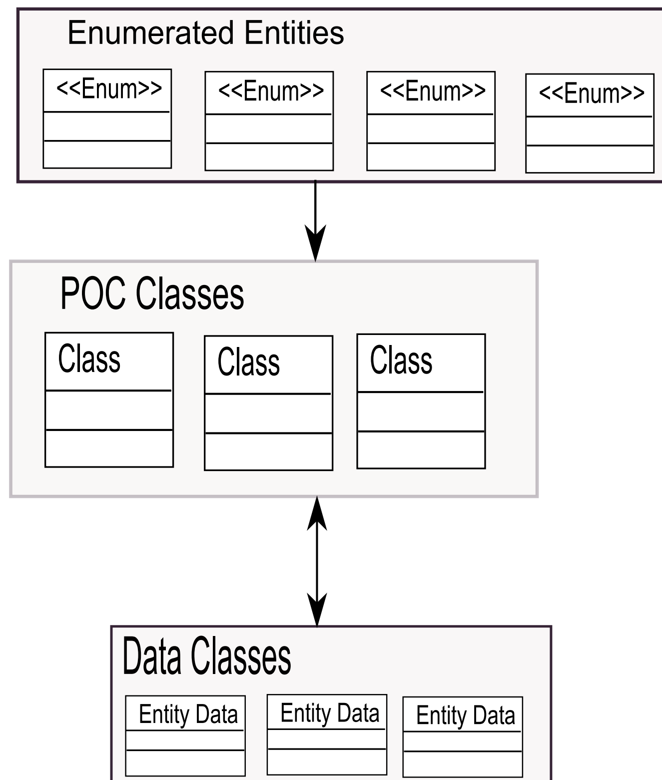

Figure 5.3: Separation of Data and Behavior

- **Data Classes.** Reading and writing of data related to domain entities. These appear in code as data classes without any associated operations. For example, the data class working on patient tubes is invoked from different component classes (cf. Figure 5.3). Thus, this data class appears as crosscutting in Aspect Browser.

- **Aspectual Types.** Aspectual types appear because crosscutting behavior related to the absence of aspects is scattered in component classes. Aspectual types provide the encapsulate this behavior and are used to interleave aspectual behavior in component classes by calling their methods. For example, tracing appears as crosscutting because it is used to invoke tracing methods to trace domain entities-related operations.

Aspect Browser (and other techniques searching for identifier crosscutting) are apt for discovering Type scattering. This is because Type scattering can generally be identified through the aggregation of scattered identifiers and Aspect Browser and similar techniques search for scattered identifiers.

Behavioral Scattering Behavioral scattering means that two distinct behaviors are composed together in a single abstraction. In our component classes, this usually happens in the form of method calls, hence indicated by abnormal high FAN-in. Following are the scenarios for behavioral scattering to occur:

- The required data is away from its behavior, therefore one behavior perpetually calls the other one to get its particular data. This results in high FAN-in value for accessors to attributes in data classes.
- Lack of a proper encapsulation for a behavior related to an entity and the behavior is *divulged* into several client classes of the entity. This causes the client classes to perpetually call the provider-logic, causing a high FAN-in value for logic-provider methods.
- A method may provide key or central information. For example, a method always passes through the patient data to get associated results and since result logic used is quite often, this results in access to patient information from all the client locations.
- Lastly, behavioral scattering occurs because a particular concern is impossible to be encapsulated in a particular abstraction using traditional OO techniques hence resulting in scattered behavioral composition of the crosscutting calls in the client locations such as caching and logging operations in our software system.

Hence in the absence of elementary OO design crosscutting "seeds" identified by the FAN-in tool consist of the following types of methods. These method types are depicted in Figure 5.4: Boxes are marked with methods or classes and arrows represent method invocations.

- **Non-Abstracted Client.** A method being called from non-abstracted, duplicated code corresponding to domain entity subtypes or from a method providing multifarious behavior as demonstrated in Figure 5.4. This is a case of missing use of polymorphism [DDN02].
- **Diverse-Logic Provider.** A method providing logic for the same entity subtypes, as depicted in Figure 5.4. This occurs because of the absence of template behavior in a parent class method or due to the presence of operation related to different types in a single method. UpdatePhysicalMeasures method described earlier belongs to this type.
- **Central Method.** A method may provide central or key information such as patient search function which is used from various other methods.
- **Divulged-Logic.** This lack of abstraction and encapsulation happens when the behavior of an object is not defined in a specific class but spread into client classes as shown in Figure 5.4.

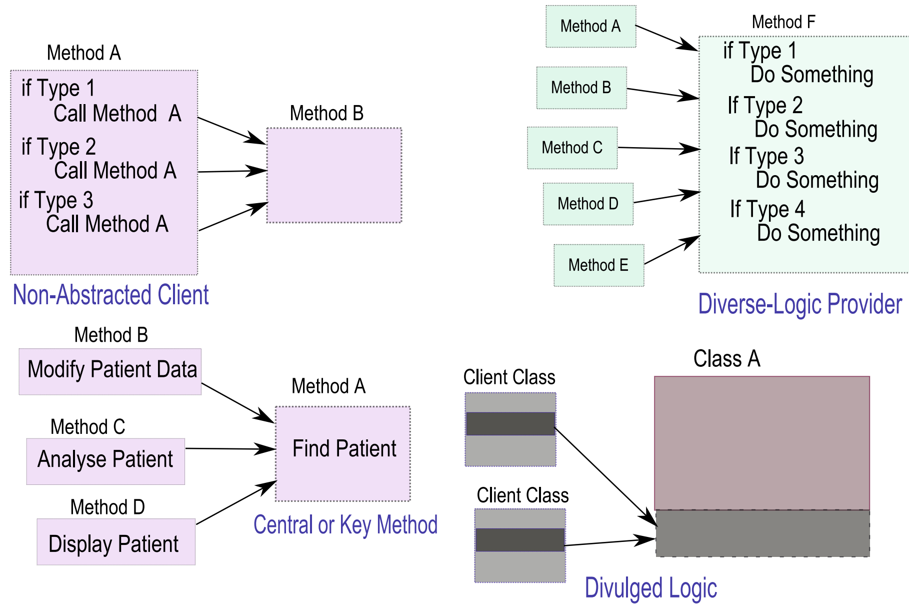

Figure 5.4: Classification of Scattered Method Calls

- **Utility Method.** A method providing utility functions such as *toString*.
- **Aspects.** A method providing scattered technical behavior such as transaction, logging, and exception handling.

It can be easily said that the scattered behavior can be detected by using Fan-in technique or other similar techniques detecting scattered behavior through method calls. These provide us a whole set of scattered behavior in code. So, we have extended the taxonomy of the scattered behavior detected by Fan-in metric in POC [BD08].

5.3 Discussion

Aspectual and domain types can be only be distinguished with the knowledge of application domain. Moreover, various scattered method calls also require domain information for their classification. For the purpose of this study, we manually applied this information to understand the results. Nonetheless, domain information cannot be acquired in an automated manner and requires human expertise to analyze aspect mining results.

However, the need for this knowledge of when inspecting aspect mining results is not novel. Roy et al. mention that the initial results from aspect mining techniques require domain knowledge to accomplish aspect mining tasks with accuracy [RURD07].

We deliberately omitted the information regarding the precision and recall of aspect mining results. We think that the purpose of this chapter is to demonstrate and reflect upon the limitations of aspect mining vis-à-vis POC. Various Studies provide information regarding accuracy of aspect mining techniques and their combinations, along with their information retrieval facts and characteristics [CMM^{+}05, RURD07]. Here we only provide a taxonomy of the results that are obtained while mining aspects in POC.

5.4 Conclusion

In this chapter, we have demonstrated that the existing aspect mining techniques produce huge number of false-positive results in the absence of the object-oriented design concepts such as inheritance and polymorphism. Hence, aspect mining results produce huge false-positives ($\sim 60\%$ of results). In the chapter, we evaluated two aspect mining tools on POC and analyzed various types of crosscutting code identified by these tools in POC. Globally, type scattering can be detected through the usage of Aspect Browser and similar techniques. Behavioral scattering can be identified through the use of Fan-in metric and similar techniques.

Hence, while mining aspects in POC, it is necessary to classify and distinguish aspect candidates from those indicating scattered domain logic. In the following chapter, we present an approach for the classification of diverse crosscutting concerns mined with aspect mining techniques by incorporating information extracted from the use of variables representing domain entities.

Chapter 6

Concern Classification in POC

A matter that becomes clear ceases to concern us.

Friedrich Wilhelm Nietzsche.

6.1 Overview

Lack of object-oriented design and procedural thinking result in scattered and tangled code related to domain entities because their data and the associated behavior do not share the same abstraction. In addition, missing class hierarchies for domain entity code also result in scattered code. Thus, crosscutting concerns also appear for *non-abstracted domain logic i.e.,* the domain entity logic that is not encapsulated in its associated class. Therefore, aspect mining techniques to discover crosscutting concerns present in a software system are inadequate in the presence of non-abstracted domain logic as they wrongly associate the lack of object-oriented structure with aspects. This occurs because scattered code is often not a sign of missing aspects but missing object-oriented abstractions. This situation introduces huge noise during aspect mining. Therefore, there is a need to understand structural difference between scattered code appearing due to the absence of objects and aspects to improve aspect mining techniques. Moreover, we need to understand the extent of various scattered concerns in programs to understand the difference between characteristics of scattering appearing due to objects and aspects.

Most of the existing publications in the domain of software restructuring for object-oriented software can be divided into four categories:

- Class hierarchy reengineering through the usage of attributes and methods of the classes making up a system and grouping those which are used together [ADN05a, Cas94, DDHL96, Moo96, SS04].

- Software refactoring through the *manual* identification of *small* design problems within class hierarchies and provides various heuristics for their rectification [FBB+99, DDN02].

- Identification of classes and objects in legacy code through the identification of common usage of data by various methods [SMLD97].
- Aspect mining to refactor crosscutting concerns in aspects [KMT07, CMM+05].

Class hierarchy reengineering, object identification and software refactoring do not target the occurrence of scattered and tangled code phenomenon due to missing abstractions. In addition, they do not take into account those concerns that can benefit from the usage of AOP mechanisms. Aspect mining techniques, as aforementioned do not distinguish amongst various crosscutting concerns those resulting from the absent OO abstractions.

In this chapter, we introduce an approach for the classification of diverse crosscutting concerns mined with aspect mining techniques in procedural object-oriented code [BDR08]. The approach classifies these concerns by taking into account domain entity data. The approach is based on the Fan-in aspect mining technique [MDM07]. One vocabulary point: In this chapter, we use the term crosscutting concerns for all the candidate concerns identified by aspect mining techniques, whether they appear due to the absence of objects or aspects.

For the approach we proceed as follows. Fan-in technique is used in the first step to mine diverse crosscutting concerns present in programs. The approach then classifies crosscutting concerns present in a software system: aspects as well as non-abstracted domain logic. Crosscutting concerns pertaining to non-abstracted domain entities are identified and extracted from the overall list of candidates through their usage of application data.

In addition, a new metric called *spread-out* is introduced to quantify the divulgence of diverse crosscutting concerns.

This chapter is organized as follows: Section 6.2 presents our approach for the identification and classification of concerns. Section 6.3 presents one new metric to quantify concern spread. Section 6.4 discusses our approach and the concern scattering metrics. Section 6.5 concludes the chapter.

6.2 Concern Classification

In the previous chapter, we have shown that the Fan-in technique can identify different types of crosscutting concerns (pertaining to the absence of aspects and objects encapsulating domain entities). But the current aspect mining techniques do not help in distinguishing the origin of various crosscutting concerns. We believe that it is important to distinguish the various kinds of crosscutting concerns so that the false-positives are filtered. Otherwise, the developers will be left to deal with too many false-positives in aspect mining candidates. For this purpose, we propose an identification for those crosscutting concerns appearing due to the lack of abstraction for domain entities so that these can be removed from the list of aspect candidates.

This section describes the approach illustrated in Figure 6.1, and is organized as follows: Section 6.2.1 defines a model for our concern classification approach and Section 6.2.2 describes the assignment of various methods to the concerns that represent domain entities. Section 6.2.3 describes the algorithm for the classification of concerns.

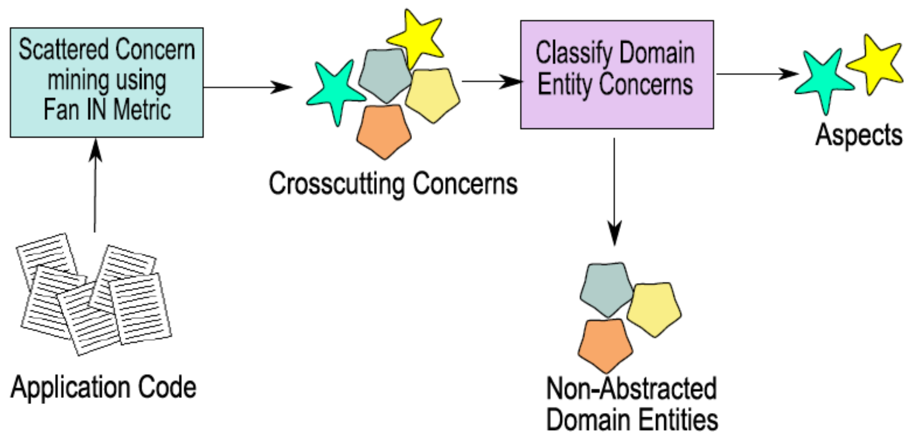

Figure 6.1: Concern Classification Approach

Scattered Methods and Domain Entities. In the previous chapter, we have extended the taxonomy of scattered method calls detected by Fan-in in POC to include those method calls that appear because of the absence of classes for domain entities. These scattered method calls appear because of the reasons described below.

- Non-abstracted methods *i.e.,* methods encapsulating operations related to different types appear because of the absence of classes and class hierarchies. These results elevated method calls for their callers and invoked methods.
- Data is placed far from its behavior. Hence, methods for reading domain entities and writing them have higher number of method calls.
- The absence of template behavior in parent classes increase the Fan-in metric for the invoked methods.

During our analysis of scattered method calls, it was revealed that these method calls were always directly or indirectly related to domain entities. That is, all scattered method calls related to false-positive candidates in Fan-in results, read or write domain entity attributes or they called methods, which read or write domain entity attributes. For example, the methods presented in Table 5.2 such as SearchPatient, UpdatePhysicalMeasures, and CreateResultCalibration read information

from data classes or write information to them. These methods have high Fan-in metric because either the data is placed far from its behavior (SearchPatient), the methods provide diverse logic related to different types (UpdatePhysicalMeasures), or missing template behavior in a parent class (CreateResultCalibration). The common property of all these false-positives candidates is that these access the data related to domain entities in the software system.

Hence, to extricate the subset of crosscutting concerns appearing due to absent domain entities, we need to identify the data and its associated behavior. Once we identify the data and behavior related to domain entities and disentangle it from the overall set of crosscutting concerns, this will remove the noise from aspect mining candidates.

6.2.1 Model for Concern Classification

To identify crosscutting concerns appearing due to the absent domain entities, we define a model based on application data usage and resolution of associated behavior. The model is based on the interaction of classes as represented in Figure 5.3 (cf. Chapter 5). In our case, data classes are classes that encapsulate data related to domain entities. Enumerated types provide the type and state information of the domain entities. The methods are encapsulated in separate POC classes that support behavior of the domain entities.

The model takes into account all the data classes, enumerated types, and methods that are present in the software system. We define M as a set of all methods in the POC class being analyzed. D is defined as a set of all objects representing data classes, and V is defined as a set of all the global variables of the enumerated types that represent the states and types for domain entities. The association amongst enumerated types and data classes is performed manually so that a domain entity is defined by a data class and one or more enumerated types.

Domain Entity Model. As previously mentioned, we assume that the data mainly consists of the representation of various domain entities in the form of data classes, and global state and type variables. Moreover, it is assumed that data classes unambiguously represent one of the domain entities *i.e.,* a data class can be considered as the representation of a single domain entity. Also, the system states and entity types represented by global enumerated types can also be associated to a single domain entity. We believe that this assumption is reasonable because all domain entities implemented in a program do have representation in code. In case of missing data classes or enumerated types related to a single domain entity, the model will not be able to associate methods and data classes to a particular domain entity.

The one-to-one association between the domain entities and the above-mentioned data components *i.e.,* variables and data classes, is utilized to determine the methods related to each domain entity. This is done by resolving the data accessed by each method.

We define E as the set of all domain entities that are implemented by the software system. Entity $e \in E$ is a combination of a data class $d(e)$ and variable $v(e)$ related to the associated domain entity e *i.e.,* $entity(e) = d(e) \vee v(e)$.

Hence, all methods in M accessing directly or indirectly domain entity-related data e are classified as implementing the concern related to the domain entity it accesses. In the example of Figure 6.2, methods of class A and class B access data “D” of class C either directly or through accessors hence they are identified as implementing concern relating to the entity “D”.

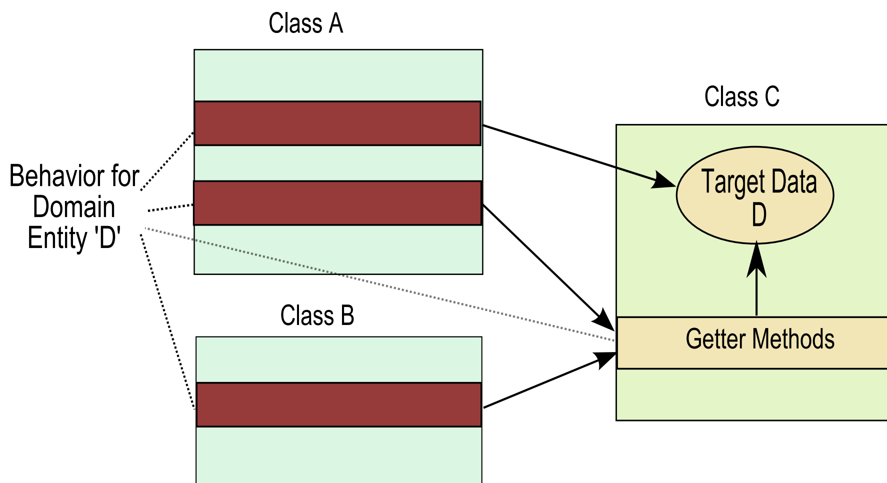

Figure 6.2: Domain Entity Concern Identification

Aspect Model. It is assumed that crosscutting concerns also appear due to the absence of appropriate OOP mechanisms to interleave two intersecting behaviors in a non-recurrent way [KLM+97]. Due to this reason, crosscutting concerns appear scattered in code. There are various forms of manifestation of crosscutting concerns in code such as cloned code, scattered method calls, and scattered identifiers [KMT07]. It is proposed that the cloned code should be refactored in helper methods for better reusability [FBB+99]. This leads to the appearance of scattered calls at places from where cloned code is replaced with method calls. Scattered identifiers are used to search for crosscutting concerns by searching types, and searching methods invoked by these types [HK01, ZJ04].

In searching for a representative technique for identification of aspects, we base our crosscutting identification model on the Fan-in metric [MDM07] (*i.e.,* the higher the number of calls to a method, the more the chances are for it being an aspects). It seems reasonable because of the fact that a comprehensive amount

of aspect mining techniques search for the occurrence of scattered and tangled method calls to detect aspects [KMT07]. Scattered method calls appear because interleaving and composition of distinct behavior in object-oriented application is performed through method calls.

For the qualification of scattered method calls as aspects, we only consider those methods which do not directly or indirectly related to domain entities. In general, such methods are invoked by those methods which are associated to domain entities as depicted in Figure 6.3. This assertion seems to have some support in literature [GK05]. In the following, a model is introduced to ascertain methods implementing domain entity related concerns are identified

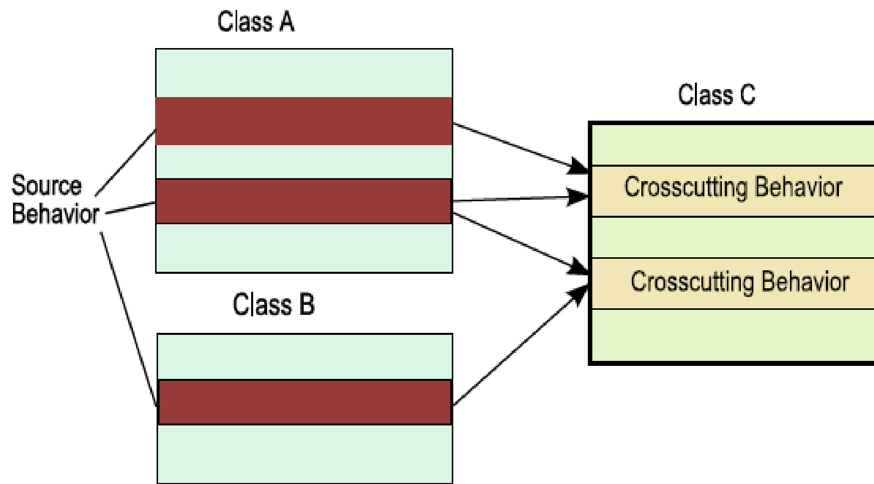

Figure 6.3: Aspect Identification

6.2.2 Domain Entity Concern Assignment

To classify methods related to domain entities, we define following primary properties based on our model.

- m *reads* d means that m directly reads from data class $d \in D$
- m *writes* d means that m directly writes to data class $d \in D$
- m *reads* v means that m directly reads from the variable $v \in V$
- m *calls* n means that m calls another method n

As variable types are read-only, we can only read from these variable types but we cannot write them.

- m *accesses* d means that m directly reads from or writes to data class $d \in D$ (*i.e., accesses* = *reads* $\cup$ *writes*)

From these primary properties, we define the following derived properties for a **concern** c.

- m *implements* c related to domain entity e if m *accesses* $d(e)$ or m *reads* $v(e)$ ***i.e.,***

$implements(m, c) = \{m \in M | \exists e \in E : m$ *accesses* $d(e) \vee m$ *reads* $v(e)\}$

- Method n implements a concern c if n calls another method m and m *implements* c pertaining to domain entity e

$implements(n, c) = \{n \in M | \exists m \in M : n$ *calls* $m \wedge implements(m, c)\}$

We do not consider the classes during the concern identification because they do not mean much in term of coherent abstraction: As stated earlier, classes in POC are partially decomposed classes without any sharp focus.

The model helps identify all the methods that are related to domain entities. The methods implementing domain entity concerns and appearing in the list of crosscutting concerns are removed from the list of crosscutting concerns. For this purpose, we describe a simple algorithm as described in the following section.

6.2.3 Algorithm for Concern Classification

We now define a simple algorithm to distinguish various crosscutting concerns discovered in the system by the Fan-in tool in Table 5.2 in the precedent chapter. The algorithm works as follow: All the crosscutting methods having a threshold value higher than f are added to the set crosscutting seeds. Each method is then examined to implement concerns related to the domain entities. Once the domain entity related methods have been marked, all the methods which are marked as crosscutting seeds and have not been marked as related to domain entities are aspect candidates.

Now that we have defined an approach for the classification of crosscutting concerns, it is also interesting to understand and examine the concerns identified so far using scattering metrics proposed in the literature. This activity may provide useful information such as the scattering patterns and extent of scattering of crosscutting concerns.

6.3 Scattering Metrics of Crosscutting Concerns

To understand the scattering of the different crosscutting concerns mined with aspect mining techniques in POC, we examine them using the existing concern scattering quantification metrics [EZS+08, KSG+06]. In addition, we describe a new metric called Spread-out to measure the divulgence of a concern over other classes.

1. $\{M\} \leftarrow \forall$ *Methods*
2. *Test all* $m \in \{M\}$ *for a Fan-in metric f*
3. *if* $fanin(m) > f$
4. $\{CCSeeds\} \leftarrow m$
5. $\forall m \in \{M\}$ = *Iterate over Instructions of m*
6. *if* $implements(m, c)$
7. $\{$*Domain Concern*$\} \leftarrow m$
8. $M \leftarrow M/m$ $\{$*Remove m from* $M\}$
9. *if* $n \in \{M \cap CCSeeds\}$
10. $\{CC\} \leftarrow n$

Figure 6.4: Concern Classification Algorithm

The overall purpose of this exercise is to understand the scattering characteristics of diverse crosscutting concerns mined in POC.

We shall adopt the model defined in Section 6.2.2 to calculate existing concern scattering metrics and our new metric. The existing metrics are redefined to calculate these metrics in an automated manner.

Three metrics are defined to quantify the dispersion of various artifacts over application classes: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO) and concern diffusion over Lines of Code (CDLOC) [KSG+06]. These metrics respectively quantify the number of classes associated with a concern (CDC), the number of methods associated with a concern (CDO), and lines of code associated with a concern (CDLOC). Since we tag methods with related domain entities, and hence concerns, we employ the CDO metric to understand the scattering of crosscutting concerns. Concern diffusion over operations (CDO) counts the number of methods whose main purpose is to contribute to the implementation of a concern. Hence, in our case, CDO for a concern c is:

$$CDO(c) = \textit{number of implements(m,c)}$$

Eaddy et al. argued that CDO and other metrics from the same suite only discern the presence of scattering but not their extent [EZS+08]. They proposed Concentration (CONC) and DOS. CONC calculates the ratio of the number of methods implementing a particular concern in a class to the total number of methods contributing to the implementation of the concern. Concern extent is defined by Degree of Scattering (DOS) [EZS+08]. DOS is a measure of the statistical variance of the concentration of a concern over all program elements. However, originally, DOS is calculated by manually marking each line of code for a particular concern. The manual concern calculation is a laborious task for large software systems. We redefine the Concentration (CONC) and DOS metrics defined in [EZS+08]. First, we automate the task of concern calculation by associating it with before mentioned concern assignment model. Second, we include only methods, instead of

lines of code, to calculate these metrics for the crosscutting concerns. We relate the concentration to our model in the way described below.

$$CONC(c,t) = \frac{\textit{Number of implement}(m,c) \textit{ in class } t}{\textit{Total implement}(m,c)}$$

CONC calculates the ratio of the number of methods implementing a particular concern in a class to the total number of methods contributing to the implementation of the concern.

$$DOS(c) = 1 - \frac{|T| \sum_t^T (CONC(c,t) - 1/T)^2}{|T| - 1}$$

DOS is a measure of the variance of the concentration of a concern over all components with respect to the worst case (*i.e.,*, when the concern is equally scattered across all classes). DOS values can lie between 0 and 1: 0 indicates that a concern is completely localized whereas 1 indicates uniform distribution of a concern over all classes.

We intend to discover the spread of each concern, *i.e.,* how many methods implement a concern outside the concern's own class. To show this spread of each concern over other classes, we introduce a metric called *Spread-Out*. It represents the ratio of the operations, associated or contributing to a given concern, located outside the main class implementing the logic for the concern. For example, Spread-Out for the Patient Records concern identifies the ratio of the methods implementing Patient Records outside the class CPatient. This helps us discern the dispersion for that concern over other program classes. Note that there are some concerns which do not have a dedicated class associated with them; in such a case we consider the class containing the largest number of operations associated to that concern as its home location. We define $spread\text{-}out(c)$ of a concern c as follow:

$$spread\text{-}out(c) = \frac{implement(m,c) \textit{ outside principal class}}{\textit{Total implement}(m,c)}$$

We shall describe the results of the classification approach and the metrics described in this chapter in Chapter 7.

6.4 Discussion

Usage of Domain Entities and Concerns. We believe that the usage of domain entities can be useful for the classification of scattered domain logic. We believe that data attributes related to domain entities of software system are key to automatically mine concerns in a program. At least one more instance of variable usage for concern identification is presented in an approach using data-flow analysis [Tri08]. The approach searches for information sinks: variables that are used to store results. As the author claims, these represent key results related to functional

requirements of a program. Hence, information sinks along with all the variables used for calculation of the results stored in information sinks are called "concern skeleton". However, behavior is not taken into account while identifying concerns.

The domain-related attributes can provide a starting point to relate their associated behavior to particular concerns. Moreover, they can provide missing program-related semantic information for current aspect mining techniques [MKK08].

Information used. For the concern extraction activity, we do not consider it necessary to include statement-level local information because there may be certain elements such as temporary variable assignments which may not be required: A higher level abstraction of the program is more useful [RM02]. However, in our case classes are non-cohesive units that do not represent useful information for concern extraction. Thus, we include only methods and global variables.

Threats to External Validity. One of the limitations of our work is the fact that it bases on the model of method invocation and Fan-in metric which assumes that there is a minimum of behavioral encapsulation in the form of methods and these methods represent a well-defined, crisp functionality. In situations where there is a haphazard, extensive scattering *i.e.,* methods do not have sharp focus and data components do not have accessors, in such cases this approach will not produce any meaningful crosscutting candidates. Crosscutting can also occur in the form of code idioms to give rise to code clones [BvDTvE04], which Fan-in wouldn't detect and hence possible combination of clones classes and domain entity data has to be combined to adapt the approach.

We also suppose that programs generally represent domain entities through well defined, succinct global variables, which help to relate methods with concerns. In the absence of such variables, a manual effort is required to associate methods with concerns. In general, automatically identifying and classifying crosscutting concerns out of completely unstructured code is near to impossible, if not impossible because in such cases there is no anchor point to start the automatic search for possible candidates. In such scenarios, Dynamic Analysis techniques can be helpful to locate crosscutting features and concerns [EKS03].

6.5 Conclusion

Crosscutting concerns may appear due to non-abstracted domain logic as well as due to the shortcomings of OO mechanisms to capture inherent crosscutting of concerns. Aspect mining techniques are capable of identifying diverse crosscutting concerns but are not capable to distinguish between them. In this chapter, crosscutting concerns originating from non-abstracted domain logic are identified according to their association to domain entities. Crosscutting concerns related to absent domain entities are identified because they access domain entity data. We

have also studied the existing metrics to quantify the concern scattering and introduced a new metric called Spread-out to support results from the classification approach. The metric helps discern the amount of behavior of a concern that has been divulged in client classes. To the best of our knowledge, the approach presented in the chapter is the first one towards the distinction of diverse crosscutting concerns present in a software system originating from the lack of elementary OO design and absence of aspects.

Chapter 7

Tools and Validation

In this chapter, we describe the tool support and the validation of the proposed approaches described in the dissertation. This chapter is organized as follows. Section 7.1 describes the tool support and validation for Scattering Analyzer approach. Section 7.2 describes the tool support and validation for the approach to restructure classes in POC. Section 7.3.1 describes the validation for the concern classification approach. Section 7.4 concludes the chapter.

7.1 Scattering Analyzer

We have proposed Scattering Analyzer approach to detect scattered code in POC (cf. Chapter 3). The technique integrates identifier analysis and Fan-in metric for detecting the scattered code.

We intended to validate the approach on our case study software. The absence of tool support for Fan-in metric and identifier analysis for types for C# language led us to develop a tool for analyzing this language. We build our own tool to identify the scattered identifiers and Fan-in metric for each method.

This tool actually performs static analysis on the bytecode of .NET assemblies *i.e.,* Common Intermediate Representation (CIL) [Int06]. First, this choice was retained for the ease of development since the bytecode is represented by a fixed number of IL commands. The bytecode strips actual names of variables in code and these variables are present as various types. Second, the tool is more generic as it does not depend on any particular language for its working: Any language that can compile its code into CIL representation can be analyzed. The simplicity of the bytecode also led us to develop swiftly the tool since no complex token-based analysis and abstract tree construction was required to represent the code in question. Third, the bytecode represents the executable instructions of a program, which represent a barebone skeleton of the program in question. Hence, it is much more accurate in terms of identifier identification in the code than those working with textual code information: Textual tools, sometimes, take into account comments and they are cluttered with the information regarding keywords, which are

used in the language such as *return* and *new*. For the degree of scattering, tool users can also define a value to filter out results with an inferior value. The tool also contains a provision to analyze user-defined types only. Figure 3.7 (cf. Chapter 3) represents our tool for C# assemblies that performs Fan-in analysis and identifier analysis to calculate each identifier and its spread through various classes. We discuss these two features of our tool in the next subsections.

7.1.1 Identifier Analysis

Identifier analysis uses a simple algorithm for finding the scattered identifiers in the code: It looks for all the identifiers and searches for those that occur frequently. Identifiers include the following: locals, enumerated types, classes, delegates, parameters, methods, and fields. Names for these types do not matter because bytecode strips off their names, only types are retained. The developer or maintainer can study these identifiers for problem detection. It provides for an identifier:

- aggregate of the frequency of occurrence of an identifier in the code;
- for each identifier, it displays the class and the associated method which accesses this identifier;
- search for particular identifier in the bytecode.

These features help decern the scattered code. For example, the global enumerated types are detected using this tool. The scattering information is equally important because one identifier, if encapsulated well, will only be accessed from a single class.

7.1.2 Identifier Results

Figure 3.7 provides a snapshot of our tool. The tool displays the enumerated type used to replace the absence of various types of tests in the system and their frequency of appearance along with classes accessing this identifier.

Table 7.1 contains the results obtained with the tool while searching for various identifiers in our case study software. For precision, we only present the results related to the enumerated types discovered through the tool. The table describes two properties for each enumerated type: *Frequency* describes the number of times the tool encountered the type. The column *Distinct Methods* in Table 7.1 shows the number of distinct methods where these enumerated types appeared. This metric depicts the extent of scattering of enumerated types. Some of the enumerated types are scattered across a large number of methods, like UnitType and TestState. The last column in the table also demonstrates that manually searching for these enumerated types can be laborious.

In addition, the identifier can also help decern the duplicate methods that reside in different classes. But these methods perform same operations on data residing in data classes. Table 7.2 shows the methods found with similar names. There are

Table 7.1: Detecting Scattered Enumerated Types

Identifier	Frequency	Distinct Methods
DossierState	11	6
ContainerType	12	6
DeterminationType	16	8
ProductType	17	8
CalibrationMode	22	10
TubeType	23	13
MethodologyType	27	12
UnitType	77	25
TestState	112	17

Table 7.2: Duplicate Methods in Identifier Analyzer

Identifier	Duplicate Methods
ModifyCQ	2
ReadMethodoType	2
ModifCalibration	2
ModifyChrono	2
ReadDetermination	2

5 method-pairs that have same names and implementing the same functionality but residing in different classes. Hence, similar names can provide quick overview for methods implementing duplicate logic. This quick overview may not be directly visible in clone detection tools.

Identifier analyzer helps providing useful information for detection of enumerated types as well as their scatter across application methods, and information regarding duplicate methods. Identifier tool provides a technique, albeit primitive one, to search for scattered identifiers. The reengineer is required to search identifiers with different frequencies to filter useless results because the aggregates depend upon the size of the system under study.

7.1.3 Fan-in Metric

The Fan-in metric, i.e. total number of calls for all methods, can be used to uncover the refactoring of duplicate code in helper methods. The tool that we developed looks for method calls to particular functions and lists those with higher values than the filtering value prescribed by the user for the degree of their scattering.

7.1.4 Fan-in Results

We detected one instance of template method pattern described in Section 3.4 with our tool. This template pattern is used in the methods that create different cali-

bration types. The calibration types are created using the same process. Thus, the methods for creating three different calibration types call similar methods.

Table 7.3: Fan-in for Glossary

Class Name	% Calls
CPatient	4%
CResult	1%
CTest	3%
CProduct	2%

In addition, with Fan-in metric, we identified the occurrence of scattered method calls related to scattered type information. In our case study software, glossary class provides the string representation of various enumerated types. Table 7.3 demonstrates the percentage calls to glossary methods in various classes and the dependence of these classes on glossary-related functionality. The glossary-related functionality is scattered in the application classes because of the absence of object-oriented design.

7.1.5 Discussion

Scattering Analyzer tool helps uncover the design defects and code smells in POC that result in scattered code. Global enumerated types are identified by the tool in the methods of the software system. It also identified scattered method calls through the use of Fan-in metric.

The Fan-in metric also helped identify other scattered methods for which we have presented a taxonomy in Chapter 5. Hence, all scattered method types are included in the Fan-in results. The reengineer is required to analyze manually each candidate to decern the reason for the high Fan-in metric of the method. The tool can be enriched with additional information such as checking method calls and associated conditionals for diverse-logic providers.

The threshold for searching for scattered identifiers and method calls is dependent upon the software system size. We do not have sufficient data to relate the threshold values with system sizes. The reengineers are required to try different values for their particular system to identify scattered code.

7.2 Reconsidering Classes: Application of the Approach

We have presented an approach for restructuring classes in POC (cf. Chapter 4). Now, we evaluate our approach on our case study software related to the software system that drives blood disease analyses. Table 1.1 demonstrates some facts about the case study software: There is a clear lack of hierarchical structure and presence of huge service classes lacking cohesion, with large number of methods. Certain

domain entities such as patient tube do not have associated classes which could have encapsulated the state and behavior related to these entities in a single class. Therefore, it offers a good case study to evaluate our approach since the software is developed in C# and is a mixture of object-oriented code and procedural one expressed within the same paradigm.

7.2.1 Tool Support

To support restructuring of POC classes, we developed a tool that performs static analysis on the bytecode of .NET assemblies *i.e.,* Common Intermediate Representation (CIL) [Int06]. The tool looks for all user-defined types and associates methods to the types to constitute principal classes according to the association rules defined in Section 4.4.1. Figure 7.1 provides a snapshot of the tool. The tool provides option to exclude namespaces, types, and methods when analyzing a software system for principal classes.

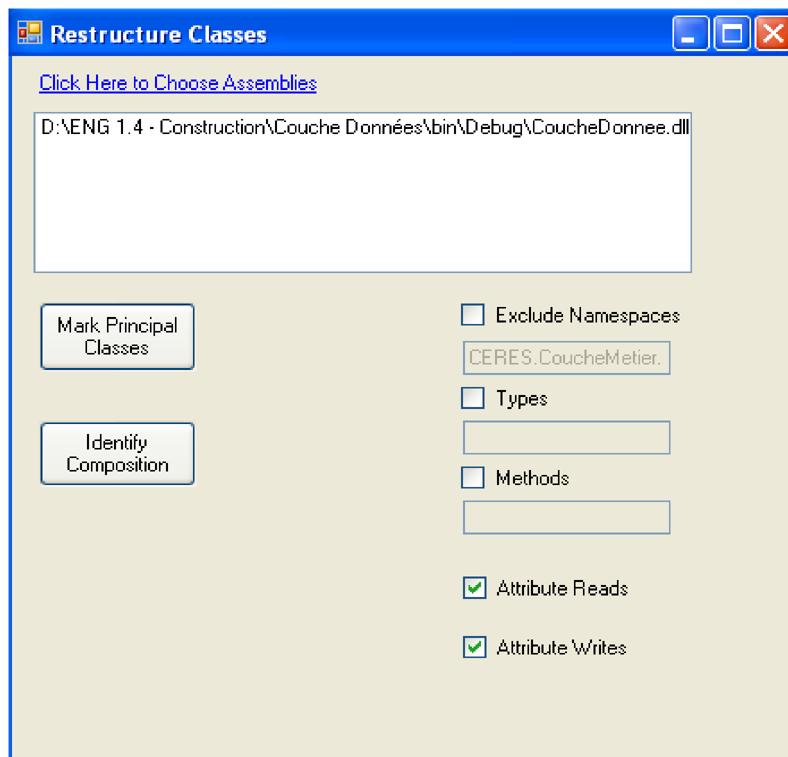

Figure 7.1: Tool for Restructuring Classes

Once, principal classes are identified, the composition links are identified. For this purpose, we look into the methods where a new object is created. If such a method is found, we look into the its invoked methods to look for the creation of new objects. Only invoked methods are searched and their invoked methods are

not taken into account *i.e.,* the callgraph of the method is searched upto a depth of 1 for new object creations. When a principal class is found to create another principal class, a composition link is created amongst the principal classes. For common compositions, a context is generated for compositions to be analyzed with a concept lattice.

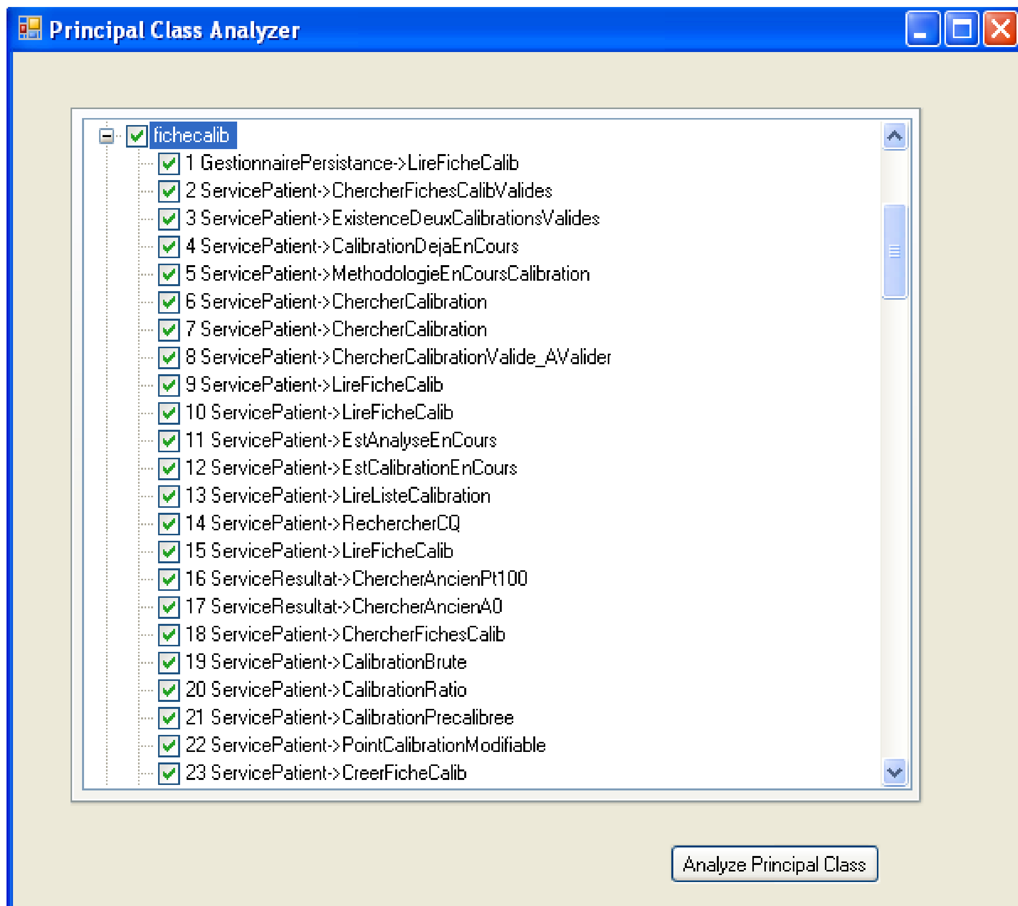

Figure 7.2: Tool for Restructuring Classes

Figure 7.2 shows a principal class and its associated methods identified by the tool. From the list of identified principal classes by the tool, the reengineer can select a principal class and analyze its associated methods to identify any false-positives associated with the principal class. The analysis mainly depends on the names of the associated methods. Otherwise, the code can be looked in for further analysis.

With the tool, the reengineer can analyze individual principal classes for extracting a class hierarchy for each of them. The tool then looks into all the methods and their access to the attributes of the current principal class. A context file is generated for the methods of the principal class with methods as objects and attributes

as attributes of the FCA context. For the purpose of concept formation, we use Galicia tool [VGRH03].

7.2.2 Validation of the Approach

Identification of Principal Classes

With the help of the tool, we proceed with the identification of principal classes within our case study software system. Following are the results of the application of the first step of our approach.

Table 7.4: Identification of Principal Classes

Total Principal Classes	41
Methods Associated Correctly	403
Methods Marked for slicing	21
False positives	7

Table 7.5: Some Principal Classes

ClassName	Method Count	ClassName	Method Count
PatientRecord	40	TestParams	21
RawResults	57	Calibration	41
InterpretedResults	9	Paramproduit	3
CalibrationData	19	PatientTube	7
QualityTest	14	BloodTest	10
TransParam	3	TransmitData	8
Product	14	Lot	11
DeviceHistory	6	Maintenance	9

Table 7.4 describes the results for the first step. We identified 41 principal classes, one for each user-defined types with the help of the tool. 403 methods were associated to these principal classes. A List of some of the principal classes along with their method count is presented in Table 7.5.

Of all the methods observed, 7 methods were determined to be false positives, that is they were associated to a class to which they didn't belong. These false-positives were identified while analyzing individually the methods associated to each of the principal class. False-positives were observed to have happened for two reasons: Firstly, when methods were reading types, the number of reads for the actual type were less than other types. For example, the method verifying if the patient information is being used in a test. Now, this information can be placed in one principal class or another depending on the proximity of data and the reengineer may be required to manually fix the position of such a method. The

second reason is more complicated as it involved the cases where a method called another method to perform operations on its type while other principal classes are accessed directly. For example, a method for repeating test on patient data called another method to recreate test information while result information was directly created in the method. This is shown in Listing 7.1 where analysis information is create by a method call while result information is created in the method. In such situation, the method is wrongly associated to the result principal class.

```
1 //code creates a new test
2 Analysis analysis = CreateNewTest();
3 Result result = new Result();
4 result.type = double;
5 result.test  = analysis;
```

Listing 7.1: Code for Creating New Analysis

There are no false-negatives of the approach because all of the methods are associated to a principal class following the association rules.

Principal Class Compositions

The creation pattern provided useful information regarding the composition of principal classes. All in all 17 composition relationships were found of which three composition relationships were identified as common to 9 principal classes.

Table 7.6: Identification of Compositions

Composite Classes	Used Classes
Calibration, Analysis, QualityControl	Raw Results
CalibProduct, Reagent, ClientProduct	Lot
CalibrationRatio, CalibrationRaw, PreCalibration	CalibProduct
ResultParams	CinetiqueDO, Cinetique2Pts, Chronometry
PatientRecord	PatientTube
Drawers	Products
TransmissionData	TransmissionParams

Table 7.6 provides a detail of the results obtained in the case study software for composition relationships. Composite classes represent those classes that contain an object of another type while used classes are contained in the composite classes. The first three rows represent composition relationships related to common superclasses where Raw Results, Lot, and CalibProduct represent the common composition relationship. Hence, three superclasses were identified with such a pattern.

Figure 7.3 illustrates an improvement in the overall design of the application because of the identification of a common composition relationship. The principal classes for three types of tests, namely, Calibration, Analysis, and QualityControl,

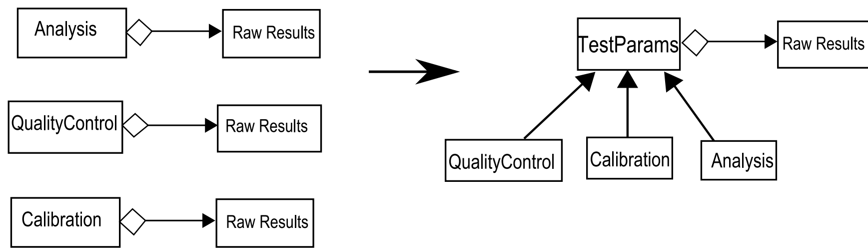

Figure 7.3: Principal Class Compositions

are composed of Raw Results. This relationship is illustrated on the left hand side of the figure. The improvement in the design of the principal classes is shown on the right hand side whereby a new class TestParams is created that abstracts the common logic for the three types of tests *i.e.,* Raw Results. Hence, common composition pattern helps to identify classes that aid in optimal usage of class hierarchies.

Hierarchies in Principal Classes

Once the task of identification of principal classes and their composition is achieved, we proceed to identify hierarchical information present within the principal classes. There are two sets of principal classes identified for our application.

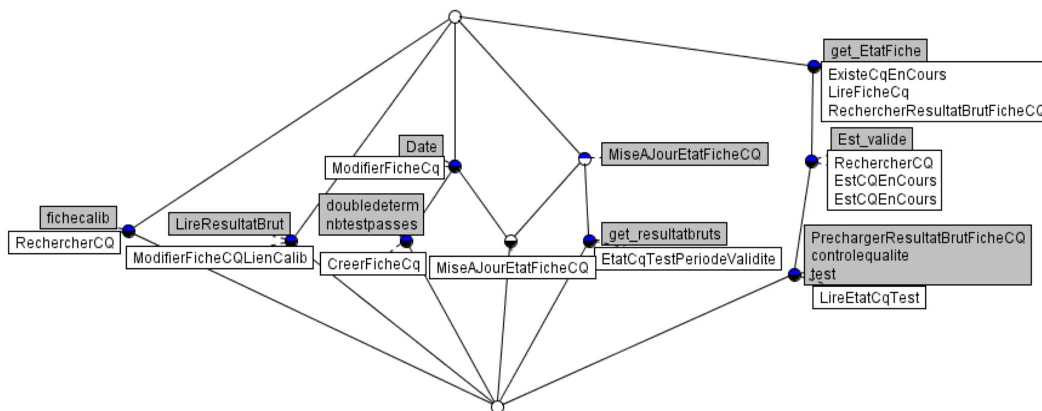

Figure 7.4: A Simple Principal Class - Quality Control

Simple Principal Classes. Simple principal classes are those for which a single concept lattice merging all the views is generated for understanding the internals of the class. This helps decerning restructuring information from a single lattice and the reengineer doesn't need to produce lattices for different views. A simple

principal class is determined from the information presented by concept lattice: If useful information can be inferred from a single lattice combining all the views, the class is termed as a simple class.

Figure 7.4 demonstrates concept lattice for QualityTest class in Table 7.5, whereby a single lattice is generated to understand the hierarchies of attribute accesses as well as method calls and type usage. The resulting concept lattice provides a clear decomposition of methods. The reengineer can further explore lattice information to create useful classes.

Complex Principal Classes. Complex principal classes are likely to consist of large number of methods. The concept lattice for such classes showing all the views contains huge number of concepts and it is near to impossible to infer any useful abstractions. Thus, for such classes subsequent views are generated to understand the hierarchies for their methods.

We consider class marked Calibration in Table 7.5. First, a fundamental view is generated to get the methods and attributes.

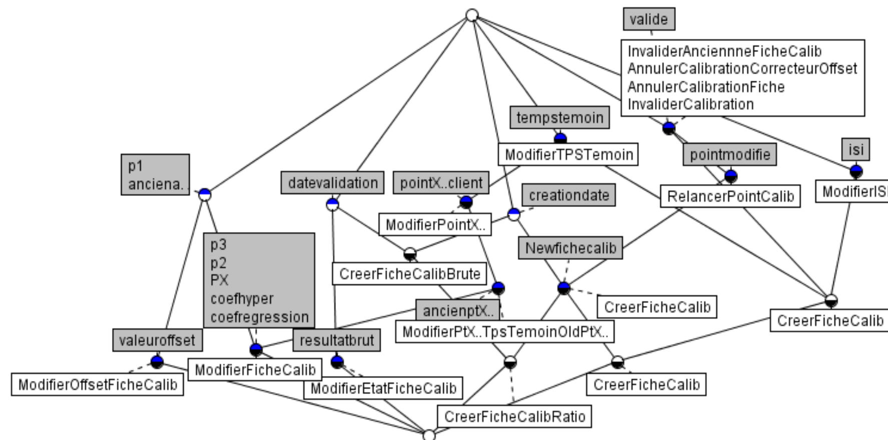

Figure 7.5: Fundamental View of a complex Class

Figure 7.5 does not reveal any particular decomposition apart from a few methods that are disjoint and can be placed in a separate class. There are strong connections amongst methods and attributes. However, the common calls view presented in Figure 7.6 does indicate presence of common calls where methods CreationCalibRatio, CreationCalibBrut, and CreationCalibPrecalibree invoke 6 common methods lying outside their principal class. These methods implement complex logic for the process of creating a calibration test, and the three methods actually implement the three types of tests. Hence a superclass is created to contain template behavior and three subclasses are then created for each process.

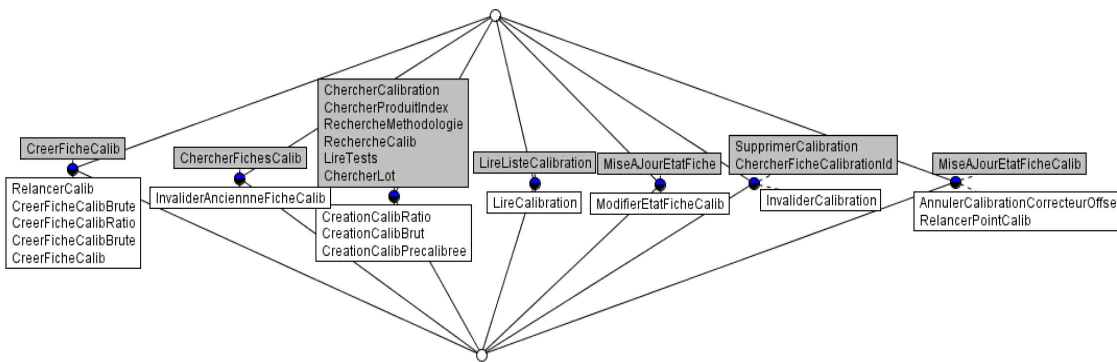

Figure 7.6: Common Interaction View of a complex Class

7.2.3 Discussion

The approach helps restructure POC by first finding cohesive set of methods and types and binding them in principal classes. In the second phase, the cohesive set are individually studied for finding useful hierarchies of methods and attributes making up a principal class. Concept lattice views help understanding the interactions of methods and attributes, methods and methods, and the interaction of the current principal class with the other principal classes to restructure POC classes in a useful way. The decomposition of the lattice information into three different views eases the inference of useful classes.

The approach is useful in that it reduces the restructuring of POC classes to the analysis of the obtained results. The overall approach reduces the overall code-level analysis of POC classes to the analysis of resulting principal classes and concept lattices, thus a few high-level design decisions. Overall the results are quite promising and validated by the developers of the case study software.

The approach is also useful in that it combines object-oriented constructs such as compositions, method calls, and association in the analysis process. The inclusion of these constructs minimizes the need to read code and infer the information manually. A complete picture of the overall software structure can be obtained from the concept lattices.

A major shortcoming of the approach lies in the fact that the reengineer needs to be proficient in understanding and interpreting the information presented by concept lattices. This information should be interpreted to form class hierarchies within principal classes. The approach does not present an algorithm to infer useful objects from the three views. We intend to study object identification algorithms in future to alleviate this shortcoming of the approach.

Moreover, the approach requires to perform various steps for restructuring purposes *i.e.,* identification of principal classes and composition relationships, and the analysis of concept lattices. The reengineer needs to combine various pieces of information obtained in the different steps of the approach to comprehend the overall

restructuring information. Our tool does not thoroughly support the process of restructuring POC classes.

However, we believe that obtaining classes and class hierarchies from an unstructured code such as POC requires a major effort that cannot be completely automated. Human intelligence is required to infer useful information from procedural object-oriented code. Human intelligence is also useful to combine domain information for inferring classes from the concept lattice views.

7.3 Classifying Crosscutting Concerns

An approach is presented for the classification of diverse crosscutting concerns identified in POC (cf. Chapter 6). Moreover, we redefined the existing set of metrics and defined a new metric for the analysis of crosscutting concerns dispersion in POC. In this section, we shall validate the classification approach and concern scattering metrics on our case study software.

7.3.1 Validating Concern Classification Approach

Table 7.7: Algorithm Results

Method	Fan-in	Classification
UpdatePhysicalMeasures	10	Domain Entity
CreateResultCalibration	10	Domain Entity
NewMeasureCalibration	10	Domain Entity
SearchProductIndex	10	Domain Entity
SearchCalib	13	Domain Entity
SearchPatient	17	Domain Entity
PublishException	19	Aspect
ReadMesureCalib	22	Domain Entity
Trace	24	Aspect
SearchProduct	26	Domain Entity
SearchTestData	29	Domain Entity
DecryptData	35	Aspect
ReadRawResults	41	Domain Entity
PublishEvent	96	Aspect
ValidateTransaction	89	Aspect
GetInstance	101	Aspect
GetGlossaryValue	127	Domain Entity

The results for the crosscutting concern classification are presented in Table 7.7. First two columns are those methods discovered as crosscutting candidates by the Fan-in tool and their corresponding Fan-in metric. The last column presents the classification for each concern obtained through the algorithm for concern classification. Candidates marked “Domain Entity” access domain entity data and are classified as the false-positives aspects.

The results produced are close to the classification that we have produced manually in Table 5.2. In addition it corresponds well with the established aspect candidates described in literature such as tracing, exception handling and transactions. Therefore, the use of domain data is useful for the classification of crosscutting concerns.

We would like to elaborate on two of the crosscutting concerns classified in Table 7.7: "GetGlossaryValue" and "DecryptData". A large number of GetGlossaryValue method calls to the glossary concern may indicate that its classification as domain entity is a false positive of the approach and that aspects may provide a better encapsulation for such a scattered concern. Had it appeared because of the missing aspects, its implementation must have benefited from the refactoring to aspects. We defined in Section 3.4 that scattered concern related to type information appear and glossary provides such an instance in our case study software. A good design will make this concern well-localized in related classes. Hence it is marked as a concern that should be inspected and rectified with object-oriented mechanism. DecryptData method call provides a service for data decryption and it can be thought of a concern working directly on domain entities. However, an analysis of encryption mechanism in our case study software reveals that domain entity data is passed to the encryption and decryption methods as parameters. Hence, data is received and returned by encryption mechanism as method arguments without directly reading domain entity data.

7.3.2 Scattering Metrics of Crosscutting Concerns

Now that we show that the use of domain data is a good indicator for classifying concerns, we want to go a step further: We study various crosscutting concerns of our software system through the light of concern scattering metrics.

The values for the scattering metrics (CDO, DOS and spread-out) for various concerns of our case study are provided in Table 7.8 (Dedicated class indicates if there is a dedicated class for the given concern).

Figure 7.7 shows the distribution for the Concern Diffusion over Operations metrics (CDO). It is described that higher values of CDO represent a concern that is more scattered [KSG+06]. In the figure, there are some entities, which do not have a dedicated class but these have higher CDO such as Calibration and Raw Results. However, Patient Tube, even in the absence of a dedicated class, has a lower CDO because it represents a smaller concern with small number of associated operations and interpreted results have higher CDO even in the presence of a dedicated class. Moreover, Interpreted Results concern has a higher CDO even in the presence of a dedicated class. Consequently, we can deduce from the figure that CDO is a relative metric which is useful to compare scattering of concerns of the same size and with the same number of associated operations. Patient Tube data and Tracing are relatively small concerns and hence, in the presence of large concerns, such as Calibration and Transactions, they tend to present smaller CDO, and hence, it is an error to infer that they are well-encapsulated. Moreover, it doesn't indicate

Table 7.8: Concern Scattering Results

Concern	Dedicated Class	CDO	DOS	Spread-out
Raw Results	No	147	0.31	0.12
Patient Records	Yes	99	0.50	0.25
Quality Control	No	39	0.55	0.21
Calibration	No	234	0.52	0.25
Analysis	No	129	0.26	0.09
Interpreted Results	Yes	228	0.68	0.43
Patient Tube	No	54	0.78	0.57
Glossary	Yes	254	0.73	0.50
Transactions	Yes	94	0.58	0.94
Tracing	Yes	26	0.47	0.93
Singleton	Yes	108	0.54	0.94
Events	Yes	110	0.57	0.88

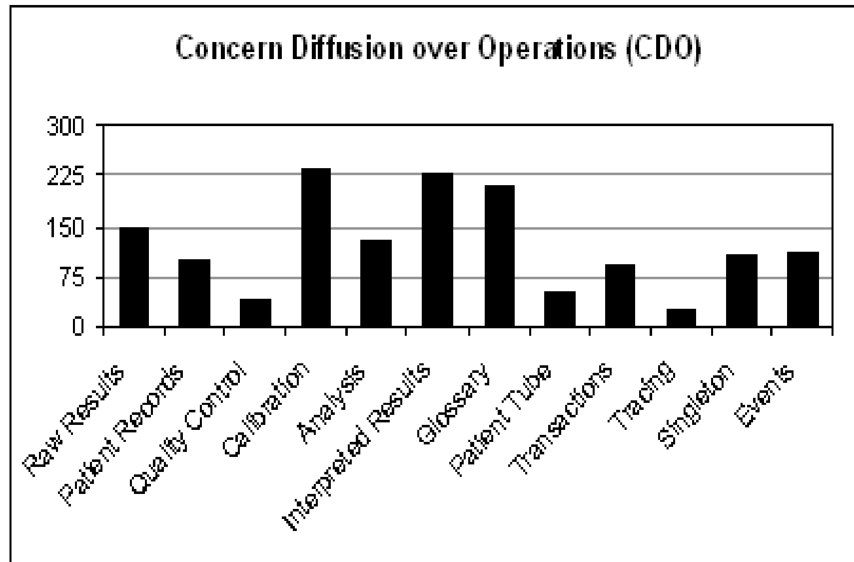

Figure 7.7: Concern Diffusion over Operations for Concerns

the extent of encapsulation for a concerns *i.e.,* whether the number of operations contributing to a concern are located inside the class implementing the concern or outside it.

Figure 7.8 shows the comparison between the DOS and Spread-out metrics; we see that all the concerns are scattered with varying degrees. Spread-out and DOS metrics also better depict the degree of scattering of the Patient Tube vis-à-vis Raw Results than CDO metric because the size of the concerns is normalized. DOS, in general demonstrates the average scattering of various concerns over classes, and all concerns have ~ 0.50 DOS. This depicts the lack of encapsu-

lation for various concerns and hence provides a measure of scattering of concerns. Spread-out demonstrates that although some of the concerns do not have an associated class, they still are concentrated in one of the subsystem classes, for example Analysis and Raw Results.

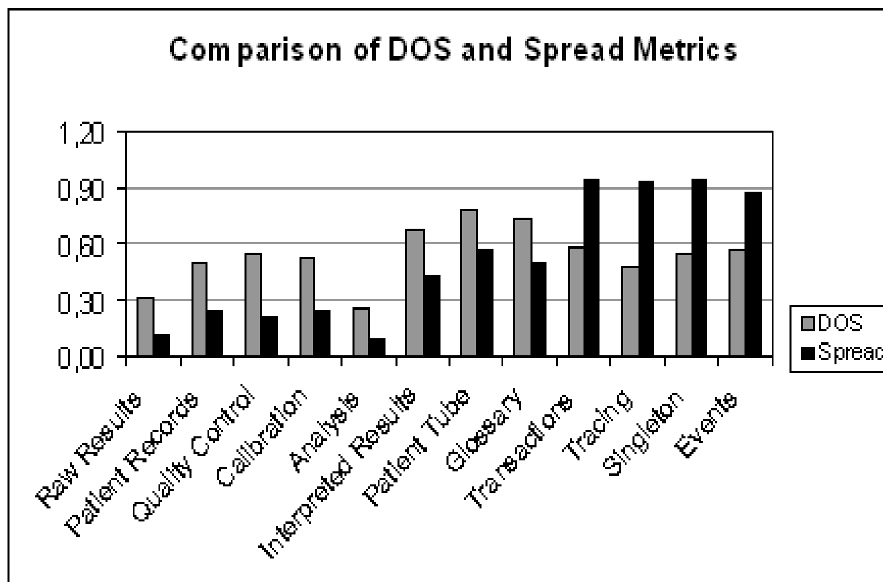

Figure 7.8: Comparison of DOS and Spread-out

Regarding the Spread-out values in Figure 7.8, it demonstrates that first that there are two kinds of Spreading-out: one part of the crosscutting concerns such as Raw Results, Patient Records, Analysis presents a low Spreading-out value. The other part of the crosscutting concerns has a high spreading-out value such as Transactions, Tracing, Singleton. This indicates two kinds of different situations which correspond well with our classification of domain entities and aspects: domain entities have lower Spread-out. On the other hand, Tracing, Singleton instances, Events and Transactions represent typical examples of aspects and their Spread-out value is much higher.

Consequently, Spread-out helps better identify the cause of crosscutting concerns than DOS and CDO: concerns exhibiting Spread-out values of more than 0.8 over the other components can be considered as aspects. And it is this anomalously extensive spread of such crosscutting concerns over other classes that one wants to capture in aspects for their improved modularity.

7.3.3 Discussion

Domain Entity Data

Our hypothesis about the domain entity usage of false-positive crosscutting concerns in general turns out to be quite substantiative because of the precision of the resulting methods that appeared in the two categories in Table 7.7. That is, most of the domain entity-related methods that were marked as crosscutting concerns have been distinguished as occurring due to lack of elementary object-oriented design.

We have evaluated our classification technique in a data-intensive program and the results are very encouraging. However, these results are needed to be validated with empirical studies. Specially, this classification algorithm needs to be evaluated in processing-intensive programs for understanding the limitation and improving the current approach.

Spread of Singleton Instances

It is striking to see the Spread-out of Singleton instances in Figure 7.8 is nearly equal to 94%. This high value of spread-out occurs because the code represents a procedural thinking therefore huge classes are written and accessed in a procedural manner. Moreover, since most of the class mostly implement unrelated concerns, the scattered concerns are composed recurrently through singleton instances, causing their spread in client classes.

Extent of Scattering

We believe that in their nature, domain entity concerns are less scattered than aspects. The reason is that domain entities are prone to collaborate with lesser number of entities. For example, patient data concern would only collaborate with patient tubes, tests, and products/reagents. On the other hand, aspects are scattering in wider because they interact with several domain entities. Aspects "crosscut" across several domain entities and hence their code is more scattered and tangled.

We have shown the higher extent of scattering of aspects with our metric Spread-out. Green et. al. [GBF+07] provide an interesting study whereby design stability for object-oriented and aspect-oriented designs of the same software is examined. Several change scenarios for software are described and evolution of design metrics is observed. They found that the changes in the crosscutting concerns in AO design are much more localized and stable than object-oriented design. It can be inferred from the provided data that the impact of changes to well-known crosscutting concerns (distribution, concurrency, exception handling) in object-oriented design have a wider impact. This impact is evident from changed components, changed operations, and changed LOC metrics [GBF+07]. However, it can equally be argued that application related changes are localized because the case study software demonstrates good design. We believe that extent of scattering should be evaluated with more empirical data.

Threats to External Validity

We have evaluated our approach on a data intensive system where we assumed that objects provide an optimal solution for scattered behavior of domain entities. We did not encounter domain entity behavior that could have benefited more from aspectual abstractions: pointcuts and advices. However, other studies that target JHotDraw, a show case system to demonstrate usage of design patterns [JHo], demonstrate Undo concern and its aspect refactoring [MDM07]. Undo concern in the program is refactored to aspects to increase modularity and this concern is directly related to domain entities: figures in JHotDraw. Such concerns will be treated as related to the absence of objects and will be considered a false-positives of our approach. Further structural analysis such as the interaction of these aspectual behavior with domain entities and their program locations can be integrated into the proposed approach to improve the identification of missing objects and aspects.

In general, our approach and spread-out metric provide encouraging results for concern classification. The approach and spread-out metric can be simultaneously employed for the classification of crosscutting concerns for better results. We believe that more empirical studies of the same sort are needed to be conducted to comprehensively understand the nature of differences between the crosscutting concerns appearing from domain entities and aspects. Our approach, nevertheless a simple one, provides a first study towards the classification of crosscutting concerns.

7.4 Conclusion

In this chapter, we have described tools for various techniques proposed in this dissertation. We have validated the techniques on our industrial case study software. Scattering Analyzer eases the detection of the scattered POC code smells. The approach for the restructuring of classes in POC aids the inference of classes and class hierarchies. However, the approach requires the analysis of concept lattices and does not propose a list of possible classes to the reengineer. The concern classification approach classifies crosscutting concerns and the results of the approach are confirmed by the spread-out metric. However, the approach requires more empirical data. The results for the approaches presented in this dissertation are quite promising. Still, the approaches need to be tested with further case studies in order to better evaluate them for their further improvement.

Chapter 8

Conclusion and Perspectives

And to make an end is to make a beginning. The end is where we start from.

Thomas Stearns Eliot.

Software reusability has a key importance for enterprises to increase return-on-investment on their software. Software reusability relies on good software design. Absence of software design or its dilution over time results in scattered code breaking the principle of modular continuity. This increases software maintenance costs and decreases software life. This dissertation described Procedural Object-Oriented Code, a kind of software developed with state of the art object-oriented languages but lacking object-oriented design. This chapter summarizes the contributions presented in the thesis and a description of the future work.

8.1 Contributions

We have discussed various research contributions in this dissertation. These are summarized below.

- *Definition of POC.* In this thesis, we have described Procedural Object-Oriented code, which appears because of the lack or erosion of overall object-oriented design. We have described a taxonomy of the design defects and code smells appearing in POC. Some of the POC design defects have similar symptoms as AntiPatterns. However, the novelty of the POC design defects is their novel manifestation in code. The design defects appearing in POC include missing classes for domain entities and the absence of class hierarchies for domain entities. These design defects give rise to certain code smells such as misplaced methods, global enumerated types, common calls, and duplicate template code. We have provided concrete examples of these code smells from our case study software. It has also been demonstrated that some of the code smells lead to scattered code, therefore rendering the software change difficult.

- *Scattering Analyzer.* Manual identification of design defects related to POC by merely looking at the lines of code can be laborious particularly because of large size of software. We have provided various existing techniques that are helpful for detecting the design defects and code smells in POC. However, certain POC code smells result in scattered code and cannot be detected with the existing techniques for the detection of design problems in object-oriented software. For this purpose, we propose Scattering Analyzer that integrates identifier analysis and Fan-in metric for the detection of the code smells that result in scattered code.

- *Reconsidering Classes in POC* A three-step approach for object-oriented design extraction from POC software is discussed. The first step consists of the identification of data and its associated behavior. This date and behavior is encapsulated in principal classes. The next step consists of the identification of relationships amongst principal classes. For this purpose, we search for the create-create patterns in code and analyze code of methods in principal classes for their associations with other principal classes. These steps are completely automated and do not require human intervention for their completion.

  Once classes and their possible associations are recovered, the next task is to understand the hierarchies of methods and attributes within principal classes. For this purpose, we utilize Formal Concept Analysis (FCA), which provides a technique to group objects having similar attributes. Three FCA views are generated with class attributes, method calls, and associations as attributes, and methods of principal classes as objects of FCA context. Moreover, we define an approach to rectify the problem of global enumerated types in POC. The approach consists of three steps: search, merge and encapsulate for global enumerated types.

  The restructuring approach can be helpful to restructure POC software into an improved object-oriented design. The approach reduces manual analysis of an entire code base into a few high-level designs. Improved design will help fulfill the promise of object-oriented design regarding reduced cost of maintenance and easier evolution.

- *Taxonomy of Scattered Code.* We have described POC to consist of partially decomposed classes. POC does not have representations for domain entities; the code related to these domain entities appear scattered in other classes of the system. Hence, aspect mining techniques that search for scattered and tangled code related to missing aspects produce false-positive in their results indicating absence of domain entities. Therefore, we evaluated two aspect mining techniques on POC and provided a list of false-positives in their results which may amount to 60% of the identified candidates. We provide a taxonomy of scattered code mined with aspect mining tools in POC. To the best of our knowledge, this work presents a first set of false-positives indi-

cating the absence of object-oriented design by aspect mining techniques. Henceforth, it would not be practical to encapsulate all candidates crosscutting concerns identified by aspect mining tools into aspects. More information needs to be incorporated in aspect mining tools to qualify crosscutting concerns as aspects.

- *Towards Concerns Classification.*

  An approach is presented based on various scattered concerns in POC to identify them in an automated manner based on the information related to domain entity types. This approach helps classify various crosscutting concerns identified by aspect mining tools. Moreover, we present a set of metrics to evaluate the scattering of crosscutting concerns. This approach helps in several ways. First, it presents an automated approach for concern identification that may reduce manual efforts required to achieve this task. For this purpose, we elaborate a simplistic model based on the notion of domain entities. Second, it helps distinguish various crosscutting concerns so that appropriate remedies can be applied to encapsulate each of their sets.

- *Validation.* Various tools are developed to provide support for the techniques proposed in this dissertation. We have validated the techniques on our industrial case study software. Scattering Analyzer eases the detection of the scattered POC code smells. The approach for the restructuring of classes in POC aids the inference of classes and class hierarchies. However, the approach requires the analysis of concept lattices and does not propose a list of possible classes to the reengineer. The concern classification approach classifies crosscutting concerns and the results of the approach are confirmed by the spread-out metric. However, the approach requires more empirical data. The results for the approaches presented in this dissertation are quite promising. Still, the approaches need to be tested with further case studies in order to better evaluate them for their further improvement.

8.2 Future Work

This dissertation presents POC and various improvements in the existing restructuring techniques to support reengineering of POC. However, we could not explore all research directions due to lack of time and lack of human resources. So, we list here several research perspective that originate from this dissertation. These are related to the improvements in the presented work.

- *Further POC Design Problems.* Absence of design results in certain design defects and code smells in POC. Existing catalogue of the POC design defects and code smells can be enriched by studying more POC systems.

- *Formalization of POC Defects.* The design defects and code smells appearing in POC are presented textually. However, their interpretations remain

subjective. These design defects and code smells can be formalized so that these can be searched, identified, and corrected in a coherent manner by automated tools.

- *Domain Entities and Scattered Code.* One of the novelties of the work presented in this work is the appearance of scattered code related to non-abstracted domain logic. We have reported a few characteristics of such code elements. This work can be carried forward to report more characteristics of scattered code appearing from non-abstracted domain logic and ways of their possible distinction from the code due to missing aspects. This distinction will consolidate the reasons for two kinds of scattered code.

- *Concern Classification.* We have presented a novel algorithm for the classification of various crosscutting concerns. Moreover, we presented the metrics for scattering. Most importantly, the algorithm and metrics require empirical validation of these should be conducted on more software systems as we believe that these are validated only on Data-intensive systems.

  Relationship of domain entities and crosscutting concerns should also be explored to understand those domain entity concerns that are best encapsulated in aspects. The study should consider the structural or behavioral differences amongst concerns that are best encapsulated with objects and aspects. This study can lead to formal definition of the structure of crosscutting concerns appearing in software.

  Concern metrics can also be integrated in the existing aspect mining techniques to include the factor of scattering to improve the search for possible aspect candidates.

- *Object Identification.*

  The object identification approach presented in this dissertation remain semi-automated and the reengineer is required to extract meaningful objects. This is because concept analysis provides structural groupings that should semantically be grouped by humans. Our future work includes definition of algorithm for moving from concepts lattices to meaningful classes. In addition, clustering algorithms are although less accurate, but they may provide a more automated alternative in this direction. So, a hybrid approach can be proposed involving both concept analysis and clustering algorithms for object identification.

- *Object and Aspect Refactoring.* This work unveils a new set of questions regarding the refactoring of code related to objects and aspects. These include, how should we go about the refactoring of code? Should we refactor objects first and aspects later or should a parallel refactoring approach be explored and developed? We would like to study these questions in detail and explore the refactoring of code into objects and aspects to measure the improvement in the overall modularity of POC.

The two perspectives can be seen as improving the identification of the application of refactoring opportunities: classification of concerns improves aspect identification, thus helping in aspect refactorings and the POC restructuring strategy provides a strategy for object-oriented refactorings. However, we do not provide a roadmap for interleaving the two kinds of refactoring opportunities together. We believe that carrying out the object-oriented and aspect-oriented refactorings and understanding their underlying relationship should form the subject of another thesis.

Appendix A

Sommaire

Les entreprises cherchent toujours les moyens de réduire le coût de développement de logiciel parce que ce coût a un effet direct sur leur compétitivité. La réutilisation des composants de logiciel participe à la réduction du coût global de son développement car les développeurs n'ont pas besoin de développer les composants à nouveau. Le logiciel réutilisable se base sur une bonne application des heuristiques de conception de logiciel, les modèles de conception, et les bonnes pratiques [GHJV95, Mey88, Rie96]. Ces modèles de modularisation de logiciel préconisent la division de grands composants dans des modules petits et autonomes. Chacun de ces modules devrait adresser une petite partie du domaine d'application et peut être évolué et maintenu indépendamment des autres.

La modularité de logiciel dépend fortement du principe de la dissimulation de l'information appelée *Information Hiding* et de l'emballage des données, appelé *Data encapsulation*. Ces principes déclarent que toutes les informations d'un module devraient être privées à moins qu'elles soient spécifiquement déclarées dans les interfaces publiques [Par72]. Ces principes assurent que la connaissance concernant un module particulier est encapsulée à l'intérieur du module. Ainsi, le changement dans les spécifications d'un système logiciel change juste un module, ou un nombre restreint de module, parce que les changements sont localisés dans les parties privées de module. Les changements du module n'effectuent pas ses interfaces publiques et par conséquent d'autres modules dépendants.

Les langages de type orienté objet fournissent le support pour la bonne modularisation de logiciel où toute la connaissance des concepts de domaine est encapsulée dans leurs classes correspondantes. Ces classes contiennent un ensemble d'opérations liées à un concept particulier de domaine. Par conséquent, le code lié à un concept de domaine, ou à une entité de domaine, est encapsulé dans sa classe particulière dans le logiciel. La fonctionnalité de ces classes est exposée à leurs clients par les interfaces bien définies. Ainsi, les clients sont inconscients des détails de l'implémentation d'une classe. Les changements dans les classes sont confinés au code résidant à l'intérieur de la classe. Par conséquent, une bonne conception orientée objet mène à la réutilisabilité de logiciel et réduit des coûts de

logiciel.

Les travaux récents sur la modularité de logiciel considèrent le logiciel comme une collection de préoccupations appelé *concerns* de logiciel [Kic96, TOHJ99]. Une préoccupation est définie en tant que "toute matière d'intérêt pour un système de logiciel" [FECA05]. On a montré que par leur nature il est difficile de modulariser quelques préoccupations de logiciel en utilisant le paradigme orienté objet et se nomment comme des *préoccupations transverses* ou *Crosscutting Concerns* [Kic96]. Des préoccupations transverses sont dispersées et embrouillées à travers des classes de l'application. Par conséquent, les préoccupations transverses violent le principe de la dissimulation de l'information et cette violation affecte sévèrement la modularité de logiciel parce que les changements de ces préoccupations transverses ne sont pas localisés dans une classe mais ils affectent des classes différentes dans le système logiciel orienté objet. La programmation orienté aspect (POA) propose d'encapsuler les préoccupations transverses [KLM+97]. Les outils et les techniques pour chercher des aspects dans les logiciels existants ont été proposés [KMT07] pour faciliter la tâche de l'identification des préoccupations transverses. Les préoccupations transverses identifiées sont ensuite encapsulées dans les aspects pour améliorer la modularité de logiciel. Pour l'identification correcte des aspects, il est important que les préoccupations transverses identifiées en code non-AOP soient correctement associées à l'absence des aspects. L'identification correcte d'aspect est essentielle pour identifier correctement les aspects candidats qui peuvent être encapsulés grâce à la POA [HRB+06].

A.1 La Problématique

Dans cette problématique, nous considérons que les logiciels utilisant des langages de type orienté objet montrent parfois une absence de conception appropriée. Ces logiciels ne peuvent pas être réutilisés, modifiés, ou maintenus sans engager des coûts élevés. Le manque de conception orienté objet se produit lorsque le processus d'analyse et de conception orienté objet est partiellement appliqué. Le manque de conception provoque la violation des principes de l'encapsulation de données et de la dissimulation de l'information dans les logiciels orientés objet. L'absence de conception orienté objet dans les logiciels ont comme conséquence certains défauts de conception et leur manifestation dans les programmes sous forme de mauvaises odeurs. En conséquence, les classes des applications sont moins cohésives et davantage couplées en raison du code dispersé, lié aux défauts de conception. Ainsi, un changement des spécifications crée des ondulations de modification dans plusieurs classes [Mey88]. D'ailleurs, le code dispersé, apparaissant à l'absence de la conception orienté objet, complique également le problème d'identification des aspects candidats.

Néanmoins, le logiciel ne peut être abandonné à cause des problèmes de conception parce que, le logiciel contient en soi la connaissance de métier et parfois il est trop cher de développer un nouveau logiciel *from scratch*. La retro-ingénierie et

la restructuration de logiciel visent à améliorer ou transformer le logiciel existant de sorte qu'il puisse être compris, changé, et réutilisé [DDN02]. En restructurant le logiciel, les défauts de conception dans le logiciel peuvent être enlevés et le logiciel peut être évolué aisément pour s'adapter au nouveau besoin [Cas98, RW98]. La restructuration de logiciel prolonge la vie du logiciel, et augmente le retour sur investissement.

Dans les sections suivantes, nous décrivons les diverses questions de recherches qui résultent de l'absence de la conception orientée objet dans un logiciel. Nous récapitulons les travaux de recherches existants pour démontrer leurs limitations vis-à-vis des questions posées dans la problématique. Plus tard, nous décrivons nos contributions qui proposent des réponses à ces questions de recherches.

A.1.1 Code orienté objet procédural

> Quelles sont les caractéristiques principales des classes et patrons dans le code qui apparat à l'absence de la conception orienté objet?

Nous appelons les logiciels développés en utilisant les langages de type orienté objet, démontrant l'absence de conception orienté objet, le code orienté objet procédural (COP). Nous croyons qu'il est important d'identifier les défauts de conception et des mauvaise odeurs de code qui apparaissent dans le COP de sorte qu'ils puissent être identifiés et détectés, et plus tard être enlevés du code. Le COP se compose de classes partiellement décomposées — les classes énormes qui définissent la logique des sous-systèmes ou services au lieu des entités particulières de domaine. Les classes partiellement décomposées apparaissent dans le COP. De ces classes partiellement décomposées dans le COP résultent certains défauts de conception au niveau de l'architecture du logiciel. Ceux-ci incluent l'absence des hiérarchies de classe. En conséquence les relations des types et des sous-types sont absents pour les entités de domaine — le code des entités de domaine n'est pas produit dans la relation hiérarchique représentée par une classe parent et ses sous-classes dérivées. Ainsi, le code lié à une entité de domaine ne peut pas être réutilisé à travers les classes dérivées. En outre, certaines entités de domaine ne sont pas représentées dans leurs classes précises, c'est-à-dire leurs propres classes sont absentes du code. Ainsi, leur code est dispersé à travers les autres classes du logiciel. La figure A.1, qui montre la graphe d'hiérarchie, démontre que souvent les classes sont des structures énormes avec une hiérarchie de classes très peu développée. Les classes énormes et l'hiérarchie de classes très peu développée mettent en évidence les classes mal décomposées et l'absence des hiérarchies de classe pour les entités de domaine présentes dans le logiciel.

Les défauts de conception dans le COP se manifestent en code comme des patrons de code, appelés généralement des mauvaises odeurs de code. Ces mauvaises odeurs de code incluent l'occurrence des appels similaires dans les méthodes, les clones de codes, les types énumérés globaux, et les méthodes mal placées. Ces odeurs de code devraient être présentées en détail, de sorte que leur manifestation

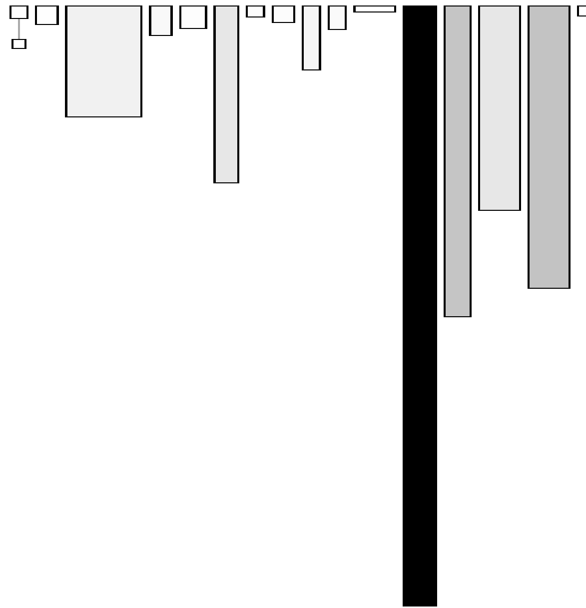

Figure A.1: *Code orienté objet procédural*

en code soit identifiée, détectée, et corrigée pour améliorer la conception orientée objet du logiciel.

Les défauts de code et de conception sont des problèmes communs et récurrents de l'implémentation et de la conception, provenant de "mauvais" choix conceptuels et qui ont pour conséquence de freiner le développement et la maintenance des logiciels en les rendant plus difficiles à maintenir et à évoluer.

Les défauts de conception sont des problèmes qui apparaissent à cause de mauvaises pratiques en matière de conception, ou les déviations des normes bien connues de conception [DM00, Rie96]. Il y a quelques travaux de recherche pertinents qui mentionnent les défauts de code et de conception dans un logiciel [BMMM98, DDN02, FBB+99, Rie96]. Cependant, les défauts de conception qui se produisent en raison de l'absence de conception orientée objet sont décrivis à niveau élevé [BMMM98]. Aucun exemple au niveau de code n'est fourni. Les odeurs de code décrivent les petites anomalies de conception de code qui représentent les opportunités de changement de code [FBB+99]. Mais les odeurs fournissent une liste d'anomalies de conception qui affectent quelques parties dans le code. Celles-ci ne fournissent pas les odeurs de code liées à l'absence de la conception orienté objet global du logiciel.

> Est il possible de fournir des techniques et des outils pour la découverte des classes de POC et les patrons de code apparaissant dans le COP?

Il y a plusieurs travaux qui traitent de l'identification des problèmes de conception dans le code. Les techniques basées sur les métriques de qualité de logiciel

et les techniques de visualisation fournissent des indices pour l'identification des classes partiellement décomposées et des hiérarchies de classes peu développées [Ciu99, LD03, Mar04, MlHG06]. Néanmoins, ceux-ci ne supportent pas l'identification des mauvaises odeurs du COP qui produisent le code dispersé à cause de l'absence de la conception orientée objet [BD07]. La détection des mauvaises odeurs du COP exige l'utilisation de l'analyse structurale des entités dispersées et de leur comportement. Par conséquent, les outils existants doivent être améliorés pour identifier correctement les mauvaises odeurs présentes dans le COP.

Le code orienté objet procédural se compose des classes, partiellement décomposées, qui encapsulent la logique concernant plusieurs entités de domaine et par conséquent une partie de ces entités de domaine n'a pas ses propres classes dans le code. Nous croyons que des défauts de conception et des odeurs de code dans le COP exercent un effet nuisible sur la modularité de logiciel et ceux-ci devraient être supprimés du logiciel. Ainsi, il est important de rechercher des manières de réorganiser les classes présentes dans le COP pour encapsuler chacune des entités de domaine dans leur classe appropriée de sorte que la modularité de logiciel COP soit améliorée.

A.1.2 La restructuration des classes dans le COP

> Pouvons-nous extraire les classes utiles orienté objet et les hiérarchies de classe à partir des classes du COP qui représentent une conception améliorée orienté objet?

On a proposé des techniques d'identification d'objet dans la littérature pour détecter des objets dans des logiciels procéduraux pour transformer ces logiciels dans la conception orienté objet [CCDD99, CCM96, NK95, SLMM99, SR99, vDK99a]. Ces techniques utilisent l'analyse formelle de concept (AFC) pour détecter des objets dans le logiciel procédural en obtenant le groupement des variables globales et des fonctions qui opèrent sur ces variables. Ces techniques cependant ne prennent pas en considération les classes partiellement décomposées présentes dans le COP et d'autres formes d'uvres orientés objet (les types énumérés et les appels de méthode) pour améliorer des algorithmes d'identification d'objet. Un autre ensemble de propositions concernant la restructuration des classes orienté objet concerne l'utilisation de AFC pour comprendre la structure interne de logiciels orientés objet et raffiner les hiérarchies de classe [ADN05a, Moo96, ST97, SS04]. Cependant, ces techniques proposent de rechercher et de corriger de petites anomalies de hiérarchie de classe dans les logiciels orientés objet. Ces approches ne visent pas de fixer des défauts de conception qui apparaissent à cause du manque de conception globale orienté objet. Ainsi, ces techniques ne sont pas aptes à transformer le COP vers une conception améliorée orientée objet.

Un autre type des propositions propose la restructuration du code orienté objet par l'identification manuelle des petits problèmes de conception dans les hiérarchies de classe et fournit les diverses heuristiques pour leur rectification [DDN02, FBB+99]. Cependant, la restructuration des classes dans le COP suivant ces derniers heuris-

tiques est trop encombrante parce que ces techniques seulement des petits problèmes de conception sans mettre la conception globale orientée objet en question.

Moha *et. al.* [MHVG08] définissent une approche très semblable au travail présenté dans cette thèse. L'approche suggère l'utilisation de l'analyse formelle de concept pour l'élimination des AntiPatterns [MHVG08] dans le code. Cependant, l'approche une fois appliquée au POC produit des treillis de concept énormes encombrés avec trop d'information qui ne peut pas être employée pour extraire de l'information utile du COP.

En résumé, il existe plusieurs approches pour l'identification d'objet dans des logiciels procéduraux, et la restructuration des logiciels et des hiérarchies de classe mais ces approches ne sont pas aptes pour la restructuration du COP vers une conception améliorée orienté objet. Ces approches une fois appliquées produisent des treillis énormes du COP qui ne permettent pas l'interprétation utile de la conception orientée objet à partir du COP. D'ailleurs, ces approches ne prennent pas en considération les mauvaises odeurs présentes dans le COP. Par conséquent, il y a un besoin de définir une approche pour restructurer des classes du COP. Cette approche devrait intégrer la structure des langages orientées objet pour obtenir la conception améliorée orienté objet du COP.

A.1.3 Identification d'aspect dans le COP

Les préoccupations transverses résultent dans la dispersion et l'embrouillement du code dans les autres préoccupations d'un logiciel. La découverte des préoccupations transverses est une tâche difficile et afin de découvrir les aspects candidats dans un logiciel, les outils et les techniques d'identification d'aspect ont été proposés [KMT07]. L'identification des aspects est une technique de rétro ingénierie. Elle automatise le processus de la découverte d'aspect dans les logiciels existants. Ces techniques se fondent sur la supposition que le code dispersé dans un logiciel est un symptôme des aspects et devrait être encapsulé dans les aspects [CMM$^+$05, HK01]. Dans les résultats, ces techniques fournissent aux utilisateurs un ou plusieurs aspects candidats *candidats* basés sur les informations lexicologiques du code, et une analyse statique ou dynamique.

Cependant, dans le cadre du COP, les techniques d'identification des aspects ne fournissent pas des résultats fiables [BD08] : Les résultats contiennent des candidats non aspects, appelé des aspects faux positifs. Le problème des aspects faux positifs se pose parce que le COP se compose du code dispersé et embrouillé non seulement dû à l'absence des aspects, mais le COP manifeste également le code dispersé apparaissant dû à l'absence de conception orientée objet. Le manque de conception orientée objet provoque le code non abstrait, c'est à dire du code lié aux entités de domaine qui n'ont pas leurs propres classes. Par conséquent, nous observons que les outils proposés pour identifier les aspects, une fois appliqué sur le COP, identifient des entités absentes de domaine dans la liste des préoccupations transverses [BD08]. Ces entités de domaine sont faussement identifiées parce que les techniques d'identification des aspects assument que la dis-

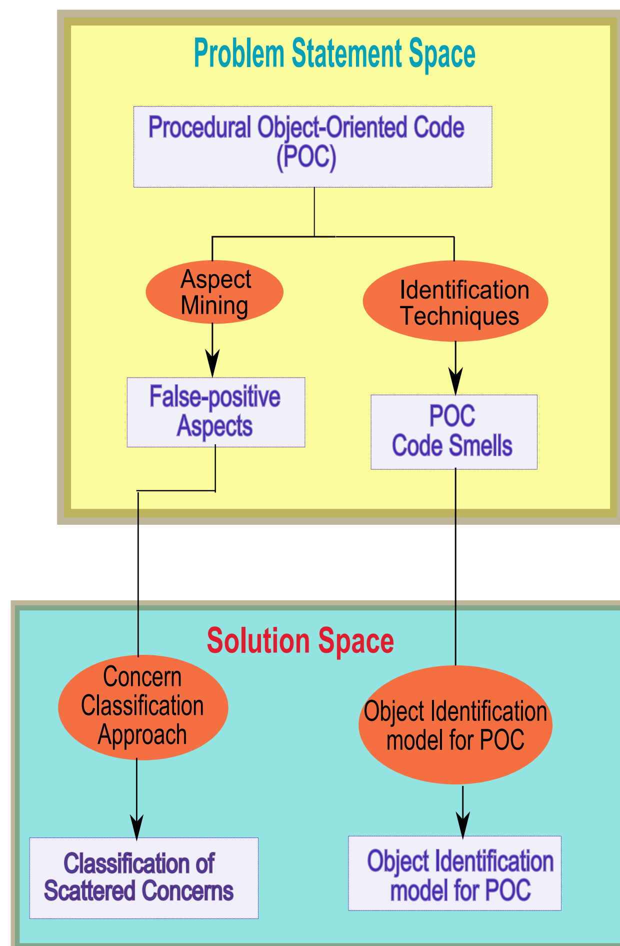

Figure A.2: Le plan de la thèse

persion et l'embrouillement proviennent seulement des aspects absents, alors que d'autres préoccupations liées aux entités de domaines ont été encapsulés dans leur abstractions orientées objet. Comme le phénomène de la bonne conception est absent dans le COP, alors nous sommes confrontés au problème de la qualification des préoccupations transverses identifiées par les outils d'identification des aspects, comme des aspects.

> Pouvons-nous différencier entre le code dispersé dans le COP pour identifier le code provenant due à l'absence des objets de la liste des aspects identifiés pour l'identification correcte des aspects?

Les techniques existantes d'identification des aspects [KMT07] ne rapportent pas des candidats faux positifs dans la liste des aspects identifiés dans le COP. Les autres travaux basés sur l'identification des préoccupations diverses présentes

dans un logiciel [EZS+08, KSG+06, RM02] fournissent des approches générales pour l'identification des préoccupations transverses sans mentionner le code dispersé apparaissant à l'absence de la conception orientée objet. Pour l'identification correcte des aspects dans le COP, il est important d'étudier les caractéristiques du code dispersé apparaissant à cause de différentes raisons. Il aidera à définir une stratégie qui classifie le code dispersé lié aux objets et aspects et nous croyons que le COP fournit une bonne occasion de classifier le code dispersé apparaissant dans les logiciels orientés objet. Cette classification aidera les techniques existantes d'identification des aspects pour différencier entre les différents types de code dispersé. Ainsi, les techniques d'identification des aspects vont mieux aider les développeurs pour distinguer le code dispersé apparaissant à cause des défauts de conception du COP et de l'absence des aspects. Cette distinction du code dispersé aide à appliquer des solutions appropriées pour encapsuler le code dispersé. L'espace global du problème de la thèse est illustré dans la figure A.2.

A.2 Contributions

Toutes les questions de recherches détaillées ci-dessus sont concernées par l'absence de conception orientée objet. Aucun travail n'indique les perspectives de restructuration des classes et de dispersion du code dans le COP. Ainsi comme mentionné brièvement dans ce chapitre, nous montrons dans cette thèse que la littérature existante ne répond pas à ces questions.

Nous avons fourni des solutions pour deux problèmes indépendants dans le COP. La première solution concerne l'occurrence du code dispersé dans le COP et sa classification. Nous choisissons cette perspective car le code dispersé provenant de l'absence des objets et des aspects dans le COP fournit une opportunité pour étudier la nature de la dispersion de code. Le but de l'approche est de proposer un perfectionnement des techniques existantes d'identification d'aspect de sorte que ces outils puissent distinguer le code dispersé lié à l'absence des aspects de celui de l'absence des objets.

La deuxième perspective est la restructuration des classes dans le COP dans le but d'améliorer la conception orientée objet d'un logiciel afin que ce dernier soit composé de classes plus cohésives et de hiérarchies de classes bien décomposées. Cette perspective est importante pour rechercher la stratégie permettant d'obtenir une conception améliorée orientée objet à partir du COP. Nous ne corrélons pas les deux perspectives présentées dans cette thèse. Nous croyons que la corrélation des deux perspectives concernant la classification de code dispersé dans le COP et la restructuration des classes devrait former le sujet pour un nouveau travail de recherche.

Dans la section ci-dessous, nous énumérons les contributions principales de la thèse. Ces contributions apportent la réponse aux questions de recherches formulées ci-dessus. Nous énumérons brièvement les contributions de cette thèse avant de fournir leur description détaillée.

- Description des défauts de conception et des mauvaises odeurs présents dans le COP avec leur stratégie d'identification. Nous définissons un outil basé sur des principes de l'identification de code dispersée pour détecter des odeurs de code dispersés dans le COP.
- Une approche pour la restructuration des classes dans le COP vers une conception améliorée orientée objet.
- Classification des préoccupations transverses trouvées dans le COP par l'analyse structurale des entités de code et les métriques.

Le chemin dans la figure A.2 illustre également les contribution de cette thèse et nous les détaillons dans les sections suivantes.

A.2.1 Mauvaise odeurs et leur détection

Dans cette thèse, nous élaborons une liste de défauts de conception et de mauvaises odeurs de code apparaissant dans le COP. Nous énumérons les défauts de conception et leurs odeurs de code associés. Les défauts de conception sont mentionnés comme les hiérarchies de classes peu développées et les classes énormes. Ces défauts de conception produisent des mauvaises odeurs dans le code. Les odeurs de code incluent des appels communs et des types énumérés globaux. Pour la détection des défauts et des odeurs dans le COP, deux groupes d'odeurs sont identifiés. Le premier groupe se compose de défauts de conception et de mauvaises odeurs de code qui démontrent les symptômes semblables aux défauts et odeurs existants. Ceux-ci peuvent être détectés par l'utilisation des métriques existantes de qualité de logiciel. Le deuxième groupe se compose des odeurs de code qui ne peuvent pas être détectées en utilisant les métriques de qualité parce que celles-ci ont comme conséquence le code dispersé. Ainsi, pour leur détection, nous avons proposé l'utilisation des techniques pour la détection du code dispersé. A cette fin, nous présentons une approche basée sur la recherche du code dispersé par l'utilisation de l'analyse d'identificateurs dans le code et de l'analyse du nombre d'appels de méthodes (*Fan-in metric*).

Le travail présenté dans cette thèse est le premier travail de recherche qui fournit un catalogue de défauts de conception et de mauvaises odeurs qui apparaissent dans le COP, et par conséquent, la première stratégie de détection pour identifier ces défauts et odeurs de code. D'ailleurs, le travail présenté ici est le premier travail qui rapporte le code dispersé résultant de l'absence de la conception orienté objet et sa stratégie de détection avec des techniques de l'identification des aspects.

A.2.2 L'approche pour la restructuration des classes

Nous présentons une approche semi-automatique et un outil pour la restructuration des classes dans le code montrant des signes d'absence de conception orientée objet [BDH08]. L'approche est basée sur l'analyse formelle de concept (AFC) [GW99].

Cette approche aide à inférer des hiérarchies de classes à grain grossier à partir des classes présentes dans le COP. L'approche globale est illustrée dans la figure A.3 et décrite ci-dessous.

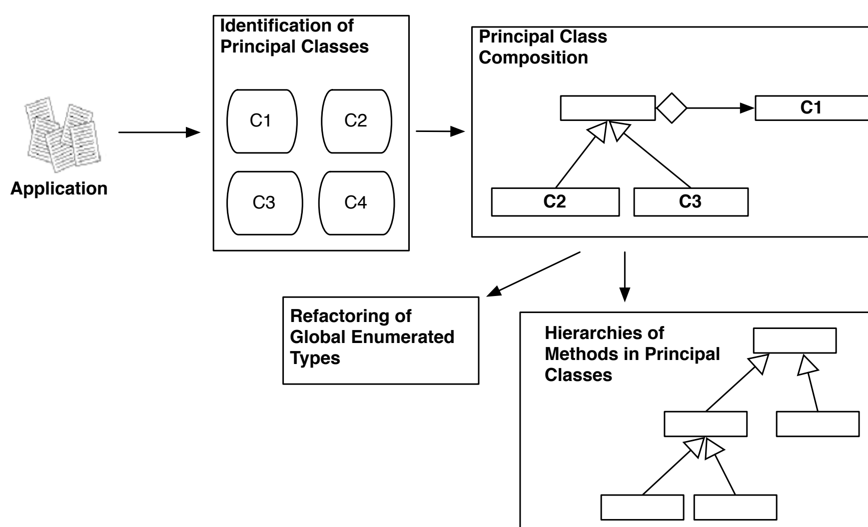

Figure A.3: Approche pour la restructuration des classes dans le COP

1. La première étape dans notre approche est la découverte des groupes cohésifs de méthodes et d'attributs dans l'application, en appliquant certaines règles, pour décomposer des grandes classes. Cette décomposition de classes est réalisée par l'analyse de l'accès des méthodes dans le code et les types définis par l'utilisateur sur lesquels les méthodes fonctionnent. Ces méthodes et types sont groupés et ces groupes cohésifs s'appellent *Principal Classes*.

2. L'abstraction architecturale orientée objet pour des classes principales est obtenue pour inférer les interactions et les compositions de classes principales entre elles. Cette abstraction architecturale est réalisée par la recherche du patron *create-create* dans le code c'est-à-dire les classes qui instancient d'autres classes.

3. Les abstractions hiérarchiques pour les méthodes et les attributs de chacune des classes principales sont obtenues par l'analyse de l'accès aux différents éléments des types associé à cette classe principale. Les abstractions sont obtenues par de divers contextes de treillis d'AFC. Nous définissons trois vues: *Fundamental View*, *Association View*, et *Common Interaction View*. Le développeur peut déduire des hiérarchies de classes pour des entités de domaine en inférant l'information hiérarchique présentée par ces vues.

4. Le code dispersé lié aux types énumérés globaux est identifié et encapsulé

dans de nouvelles méthodes. Ces méthodes sont alors ajoutées à la classe principale spécifiée par l'utilisateur.

Comme discuté auparavant, les approches existantes pour la restructuration des hiérarchies de classe ne sont pas aptes pour l'inférence des hiérarchies de classes dans le COP. Notre approche comble la lacune en produisant des treillis d'AFC pour chacune des principales classes identifiées par l'approche. La réduction de treillis d'AFC est réalisée parce que les classes principales contiennent seulement l'état et les méthodes liés à une entité particulière de domaine. Par conséquent, l'information de la classe qui est présentée dans les treillis est réduite d'une manière significative. Par conséquent, les treillis sont plus petits et il est plus facile interpréter des hiérarchies de classe.

A.2.3 L'approche pour la classification des préoccupations transverses

Nous évaluons deux techniques d'identification d'aspects sur un système industriel et nous rapportons qu'un nouvel ensemble d'aspects candidats faux positifs sont identifiés par le techniques d'identification d'aspects [SB]. Nous observons que les techniques courantes d'identification d'aspects sont insuffisantes pour distinguer le code dispersé résultant de l'absence des objets du code dispersé qui apparat à l'absence des aspects. Cette limitation des techniques d'identification d'aspects se produit parce que les techniques d'identification d'aspects lient le code dispersé aux aspects, indépendamment du fait que le code dispersé apparat à l'absence de certaines classes ou de limitations inhérentes des mécanismes orientés objet à encapsuler des préoccupations transverses. Nous décrivons brièvement une taxonomie d'outils d'identification d'aspects concernant leurs capacités de détecter des préoccupations traverses liées aux types dispersés et au comportement dispersé [BD08].

Nous proposons une approche pour classifier le code dispersé dans le COP [BDR08]. La classification adopte une approche à deux dents. Premièrement, l'approche identifie et groupe des préoccupations transverses actuelles dans un système logiciel : aspects aussi bien que entités absentes de domaine. Des préoccupations transverses concernant les entités absentes de domaine sont identifiées et extraites par leur utilisation des données d'entité de domaine. Deuxièmement, une nouvelle métrique appelé *Spread-out* est présentée pour mesurer la dispersion des diverses préoccupations transverses présentes dans un logiciel.

Notre travail est le premier travail qui rapporte l'occurrence des aspects candidats faux positifs identifiés par des techniques d'identification d'aspect dues à l'absence de la conception orientée objet [BD08]. Ainsi, l'approche de la classification présentée dans cette thèse aide les outils d'identification d'aspect à identifier et à filtrer des résultats qui apparaissent à l'absence des classes pour des entités de domaine. Le filtrage des candidats faux positifs à partir des résultats d'identification d'aspect aide dans l'encapsulation correcte des aspects.

Appendix B

Introduction to Formal Concept Analysis

B.1 Introduction

This appendix is an introduction to the main terminology of Formal Concept Analysis. It is attempted to give a global overview in this mathematical discipline to understand how FCA works, how it can be applied in different case studies. It is a summary of the definitions given in [Aré05, GW99, SR99, Sne96, ST98].

Formal Concept Analysis (FCA) [GW99](also known as Galois lattices [Wil81]) is a branch of lattice theory that allows us identify meaningful groupings of objects that have common attributes. The mathematical notations used in the following sections of this chapter are mainly extracts from *Formal Concept Analysis: Mathematical Foundations* by [GW99].

In all the extent of this report, we use one illustrative example about a crude classification of a group of mammals: cats, gibbons, dolphins, humans, and whales, and we consider five possible characteristics: four-legged, hair-covered, intelligent, marine, and thumbed. Table B.1 shows the relationships between the mammals and its characteristics.

Table B.1: Mammal example: Table T represents the binary relations

	four-legged	hair-covered	intelligent	marine	thumbed
Cats	×	×			
Dogs	×	×			
Dolphins			×	×	
Gibbons		×	×		×
Humans			×		×
Whales			×	×	

But first of all, we need to understand a few definitions to see how we analyze

the information provided by this technique. The symbols $\cap$, $\cup$, $\setminus$,$\subset$, $\subseteq$,$\in$ represent the classical operations on sets: *intersection, union, complement, belongs to, restrictive inclusion, inclusion*. The rest of the symbols that have a specific meaning in this context are introduced in the text.

B.2 Context and Concepts

The initial starting point in using FCA is setting up a **context**. A context is a triple:

$$C = (O, A, R)$$

O is a finite set of objects, A is a finite set of attributes and R is a binary relation between O and A: $R \subseteq O \times A$ and is usually represented as a cross-table T. The binary relation in our example is shown in the Table B.1, where we see that our objects are the animals and attributes are their characteristics. Then we see that the tuple *(whales, marine)* is in R but *(cats, intelligent)* is not.

Let $X \subseteq O$ and $Y \subseteq A$. The mappings:

$$\sigma(X) = a \in A | \forall o \in X : (a, o) \in R,$$

the *common properties* of X, and

$$\tau(Y) = o \in O | \forall a \in A : (a, o) \in R,$$

the *common elements* of Y , form a *Galois connection*. That is, the mappings are *antimonotone*:

$$X_1 \subseteq X_2 \rightarrow \sigma(X_2) \subseteq \sigma(X_1)$$

$$Y_1 \subseteq Y_2 \rightarrow \sigma(Y_2) \subseteq \tau(Y_1)$$

and their composition is *extensive*:

$X \in \tau(\sigma(X))$ and $Y \in \sigma(\tau(Y))$

In the mammal example:

$\sigma(\{Cats, Gibbons\}) = \{\textit{hair-covered}\}$ and $\tau(\{marine\}) = \{dolphins, whales\}$

Based on the previous definitions, we define the term of concept. A **concept** is a pair of sets: a set of elements (the *extent*) and a set of properties (the *intent*) (X, Y) such that:

$Y = \sigma(X)$ and $X = \tau(Y)$

Therefore a concept is a maximal collection of objects sharing common attributes. Informally, such a concept corresponds to a maximal rectangle in the cross-table T : any $o \in O$ has all attributes in A, and all attributes $a \in A$ fit to all objects in O. In the mammal example, $(\{Cats, Dogs\}, \{four\text{-}legged, hair\text{-}covered\})$ is a concept, whereas $(\{Cats, Gibbons\}, \{hair\text{-}covered\})$ is not a concept. Although,

$$\sigma(\{Cats, Gibbons\}) = \{hair\text{-}covered\}),$$

$$\tau(\{hair\text{-}covered\}) = \{Cats, Dogs, Gibbons\}$$

shows that it is not a concept. Table B.2 shows the complete list of concepts. It is important to note that concepts are invariant against row or column permutations in the cross-table T.

Table B.2: Concepts of the mammal example

top	(Cats, Gibbons, Dogs, Dolphins, Humans, Whales , ;)
c6	(Gibbons, Dolphins, Humans, Whales , intelligent)
c5	(Cats, Gibbons, Dogs , hair-covered)
c4	(Dolphins, Whales , intelligent, marine)
c3	(Gibbons, Humans , intelligent, thumbed)
c2	(Cats, Dogs , hair-covered, four-legged)
c1	(Gibbons , hair-covered, intelligent, thumbed)
bottom	(; , four-legged, hair-covered, intelligent, marine, thumbed)

B.3 Concept Lattice

The set of all the concepts of a given context forms a *complete partial order*. Thus we define that a concept (X_0, Y_0) is a **subconcept** of concept (X_1, Y_1), denoted by $(X_0, Y_0) \leq (X_1, Y_1)$, if $X_0 \subseteq X_1$ (or, equivalently, $Y_1 \subseteq Y_0$).

For instance, $(\{Dolphin, Whales\}, \{intelligent, marine\})$ is a subconcept of

$(\{Gibbons, Dolphins, Humans, Whales\}, \{intelligent\})$.

Thus the set of concepts constitutes a concept lattice $L(T)$. The concept lattice for the mammal example is shown in Figure B.1.

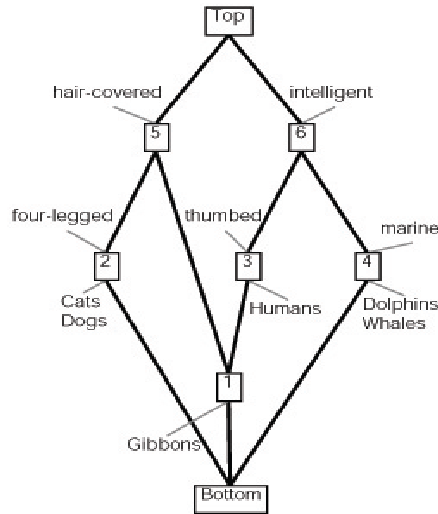

Figure B.1: The lattice of the mammals example with classical notation

Each node in the lattice represents a concept and they are shown in Table B.2. Given two elements (O_1, A_1) and (O_2, A_2) in the concept lattice, their *infimum* or *meet* is defined as:

$$(O_1, A_2) \sqcap (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2)),$$

and their *supremum* or *join* as

$$(O_1, A_2) \sqcup (O_2, A_2) = (\tau(O_1 \cap O_2), O_1 \cap O_2),$$

Following our mammal example, lets see the results of $c_3 \sqcap c_5$ and $c_1 \sqcup c_2$ when we compute them:

$$\begin{aligned} c_3 \sqcap c_5 =& (\{gibbons, humans\}, \{intelligent, thumbed\}) \\ & \sqcap (\{cats, gibbons, dogs\}, \{hair\text{-}covered\}) \\ & = (\{gibbons\}, \tau(\{gibbons\})) \\ & = (\{gibbons\}, \{hair\text{-}covered, intelligent, thumbed\}) \\ & = c1 \end{aligned}$$

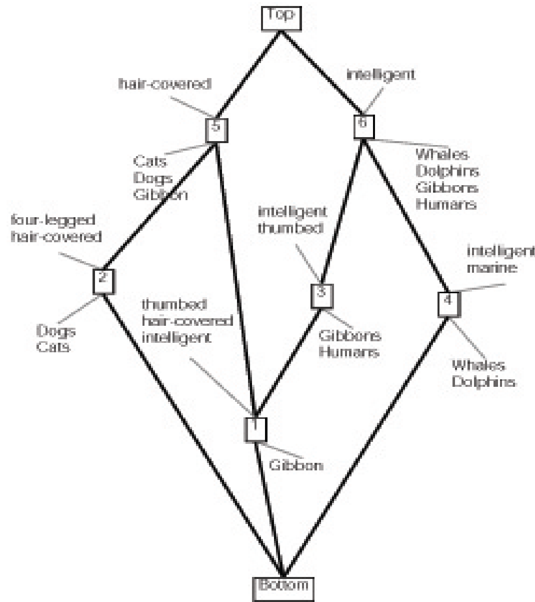

Figure B.2: The lattice of the mammals example with complete notation

$$\begin{aligned} c_1 \sqcup c_2 = & (\{gibbons\}, \{hair\text{-}covered, intelligent, thumbed\}) \\ & \sqcup (\{cats, dogs\}, \{hair\text{-}covered, four\text{-}legged\}) \\ & = (\tau(\{hair\text{-}covered\}), \{hair\text{-}covered\}) \\ & = (\{cats, dogs, gibbons\}, \{hair\text{-}covered\}) \\ & = c_5 \end{aligned}$$

Generalizing, the fundamental theorem for concept lattices [Wil81] relates sub-concepts and super-concepts as follows:

$$\bigsqcup_{i \in I} (X_i, Y_i) = \left(\tau \left(\bigcap_{i \in I} Y_i \right), \bigcap_{i \in I} Y_i \right).$$

$$\sqcap_{i \in I} (X_i, Y_i) = \left(\bigcap_{i \in I} X_i, \sigma \left(\bigcap_{i \in I} X_i \right) \right).$$

From the computation of the concepts, two *special* concepts are also introduced in the concept lattice. Given a context $C = (O, A, R)$, the two following concepts are calculated:

$$top = (\tau(\theta), \sigma(\tau(\theta)))$$

$$bottom = (\tau(\sigma(\theta)), \sigma(\theta))$$

The *top concept* reflects the attributes that fit to *all* objects, and the *bottom concept* reflects the objects that fit to *all* attributes. If there are not attributes that fit all objects and/or there are not objects that fit all attributes, the definitions are reduced to the following expressions:

$$top = (O, \theta)$$

$$bottom = (\theta, A)$$

This means that in the case of *top concept*, there is no column with crosses for all the objects in the table T ; and in the case of *bottom concept*, there is no row with crosses for all the attributes in the table T .

List of Figures

1.1 *Procedural Object-Oriented Code* — Rectangles represent classes, edges represent the inheritance relationships between classes, height and width of rectangles are dependent upon number of methods and number of attributes respectively, and node color is determined by the number of lines of code (a snapshot taken using the MOOSE reverse engineering environment [MGL06]). 4
1.2 Thesis Plan 7
1.3 An Approach for Restructuring Classes in POC 10
1.4 Testing Plasma Tubes 12

2.1 Polymetric Views: Good Design 20
2.2 Polymetric Views: Design with Defects 20
2.3 Aspect Mining and Refactoring [KMT07] 31
2.4 Aspect Mining Tools and Techniques 32

3.1 Procedural Object-Oriented Code — Rectangles represent classes, edges represent the inheritance relationships between classes, height and width of rectangles is dependent upon number of methods and number of attributes respectively, and node color is determined by the number of lines of code. 45
3.2 Taxonomy of the POC Design Defects and Code Smells 46
3.3 Arrangement of Data and Controller Classes in POC 47
3.4 Cloned Calls - Missing Template Behavior 49
3.5 Duplicate Template Code 49
3.6 Coarse-Grained Polymetric Views - Nodes represent classes, edges represent the inheritance relationships between classes, height and width of rectangles is dependent upon number of methods and number of attributes respectively, and node color is determined by the number of lines of code. 55
3.7 Scattering Analyzer 58

4.1 Subsystem Classes Access and Modify 63
4.2 Overall Object Identification Approach 65
4.3 Principal Class Identification 65

4.4 Principal Class Compositions 67
4.5 Common Compositions 68
4.6 Various Views Obtained for Class Hierarchies from Principal Classes 69
4.7 Fundamental View 70
4.8 Resulting Class Hierarchy 70
4.9 Common Calls View of Principal Classes 70
4.10 Resulting Interactions through Method Calls 71
4.11 Association View 72
4.12 Resulting Class Hierarchy 72

5.1 Scattered Concerns in Aspect Browser 81
5.2 Inter-Class and Intra-Class Method Invocations in POC 84
5.3 Separation of Data and Behavior 88
5.4 Classification of Scattered Method Calls 90

6.1 Concern Classification Approach 95
6.2 Domain Entity Concern Identification 97
6.3 Aspect Identification 98
6.4 Concern Classification Algorithm 100

7.1 Tool for Restructuring Classes 109
7.2 Tool for Restructuring Classes 110
7.3 Principal Class Compositions 113
7.4 A Simple Principal Class - Quality Control 113
7.5 Fundamental View of a complex Class 114
7.6 Common Interaction View of a complex Class 115
7.7 Concern Diffusion over Operations for Concerns 118
7.8 Comparison of DOS and Spread-out 119

A.1 *Code orienté objet procédural* 132
A.2 Le plan de la thèse 135
A.3 Approche pour la restructuration des classes dans le COP 138

B.1 The lattice of the mammals example with classical notation 144
B.2 The lattice of the mammals example with complete notation . . . 145

List of Tables

1.1 Case Study Metrics 13

5.1 Crosscutting Concerns and their Frequency 82
5.2 Application methods and associated Fan-in values 85

7.1 Detecting Scattered Enumerated Types 107
7.2 Duplicate Methods in Identifier Analyzer 107
7.3 Fan-in for Glossary 108
7.4 Identification of Principal Classes 111
7.5 Some Principal Classes 111
7.6 Identification of Compositions 112
7.7 Algorithm Results 116
7.8 Concern Scattering Results 118

B.1 Mammal example: Table T represents the binary relations 141
B.2 Concepts of the mammal example 143

Bibliography

[ADN03] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Understanding classes using X-Ray views. In *Proceedings of 2nd International Workshop on MASPEGHI 2003 (ASE 2003)*, pages 9–18. CRIM — University of Montreal (Canada), October 2003.

[ADN05a] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 62–71. IEEE Computer Society, March 2005.

[ADN05b] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Lessons learned in applying formal concept analysis. In *Proceedings of 3rd International Conference on Formal Concept Analysis (ICFCA '05)*, volume 3403 of *LNAI (Lecture Notes in Artificial Intelligence)*, pages 95–112. Springer Verlag, February 2005.

[Aré05] Gabriela Arévalo. *High Level Views in Object-Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, Berne, January 2005.

[BD07] Muhammad Usman Bhatti and Stphane Ducasse. Surgical information to detect design problems with moose. In *Proceedings of FAMOOSr (1st Workshop on FAMIX and MOOSE in Reengineering)*, 2007.

[BD08] Muhammad Usman Bhatti and Stéphane Ducasse. Mining and classification of diverse crosscutting concerns. In *LATE '08: Proceedings of the 2008 AOSD workshop on Linking aspect technology and evolution*, 2008.

[BDH08] Muhammad Usman Bhatti, Stéphane Ducasse, and Marianne Huchard. Reconsidering Classes in Procedural Object-Oriented Code. In *Proceedings of WCRE '08 (15th Working Conference on Reverse Engineering)*, pages 257–266, 2008.

[BDR08] Muhammad Usman Bhatti, Stéphane Ducasse, and Awais Rashid. Aspect mining in procedural object oriented code. In *The 16th IEEE International Conference on Program Comprehension*, pages 230–235, 2008.

[BK04] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 310–315, Washington, DC, USA, 2004. IEEE Computer Society.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[BMW94] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, May 1994.

[BNL05] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Softw. Eng.*, 31(2):137–149, 2005.

[BvDTvE04] Magiel Bruntink, Arie van Deursen, Tom Tourwe, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Washington, DC, USA, 2004. IEEE Computer Society.

[BW97] Alan W. Brown and Kurt C. Wallnau. Enginnering of component-based systems. In Alan W. Brown, editor, *Component-Based Software Engineering*, pages 7–15. IEEE Press, 1997.

[BZ06] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[Cas94] Eduardo Casais. Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, 1(2):95–115, December 1994.

[Cas98] Eduardo Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, January 1998.

[CB91] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, February 1991.

[CCDD99] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. A Case Study of Applying an Eclectic Approach to Identify Objects in Code. In *Proceedings of IWPC '99 (7th International Workshop on Program Comprehension)*, pages 136–143. IEEE, IEEE Computer Society, May 1999.

[CCM96] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Softw. Pract. Exper.*, 26(1):25–48, 1996.

[Che04] Checkstyle, 2004. `http://checkstyle.sourceforge.net/`.

[Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.

[CLLF99] Aniello Cimitile, Andrea De Lucia, Guiseppe Antonio Di Lucca, and Anna Rita Fasolino. Identifying objects in legacy systems using design metrics. *J. Syst. Softw.*, 44(3):199–211, 1999.

[CMM+05] Mario Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwe. A qualitative comparison of three aspect mining techniques. In *13th International Workshop on Program Comprehension (IWPC)*, pages 13–22. IEEE CS, 2005.

[DDHL96] Hervé Dicky, Christoph Dony, Marianne Huchard, and Thérèse Libourel. On Automatic Class Insertion with Overloading. In *Proceedings of OOPSLA '96 (11th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications)*, pages 251–267. ACM Press, 1996.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DG03] Uri Dekel and Yossi Gil. Revealing class structure with concept lattices. In *WCRE*, pages 353–362. IEEE Press, November 2003.

[DGN05] Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, September 2005. Tool demo.

[DHHaV04] Michel Dao, Marianne Huchard, Mohamed Rouane Hacene, and Cyril Roume andPetko Valtchev. Improving Generalization Level in UML Models Iterative Cross Generalization in Practice. In *Proceedings of ICCS '94 (12th International Conference on Conceptual Structures)*, volume 3127 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, July 2004.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DLT01] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, August 2001.

[DM00] et al. Didier Martin, Mark Birbeck. *Professional XML*. Wrox Press Ltd., 2000.

[DSP08] Karim Dhambri, Houari A. Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In *12th European Conference on Software Maintenance and Reengineering 2008*, pages 279–283, April 2008.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.

[ESEE92] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[EZS+08] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 2008.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[FSG+08] Eduardo Figueiredo, Cláudio Sant'Anna, Alessandro Garcia, Thiago T. Bartolomei, Walter Cazzola, and Alessandro Marchetto. On the maintainability of aspect-oriented software: A concern-oriented measurement framework. In *Conference on Software Maintenance and Reengineering, 2008*, pages 183–192, 2008.

[FxC06] Fxcop, June 2006. `http://msdn.microsoft.com/en-us/library/bb429476.aspx`.

[GBF+07] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Alessandro Garcia, Nlio Cacho, Claudio SantAnna, Paulo Borba, Uirakulesza, and Awais Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *Proceedings of ECOOP2007*, LNCS, pages 176–200. Springer-Verlag, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[GK95] Harald Gall and René Klosch. Finding objects in procedural programs: an alternative approach. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 208, Washington, DC, USA, 1995. IEEE Computer Society.

[GK05] K. Gybels and A. Kellens. Experiences with identifying aspects in Smalltalk using 'unique methods'. In *Workshop on Linking Aspect Technology and Evolution*, 2005.

[GKY00] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, 3, 2000.

[GL91] Keith Brian Gallagher and James R. Lyle. Using Program Slicing in Software Maintenance. *Transactions on Software Engineering*, 17(18):751–761, August 1991.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[Har00] Maarit Harsu. Identifying object-oriented features from procedural software. *Nordic Journal of Computing*, 7(2):126–142, 2000.

[HK01] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition in legacy code. 2001.

[HOU] Stefan Hanenberg, Christian Oberschulte, and Rainer Unl. Refactoring of aspect-oriented software. In *Proceedings Net.ObjectDays, year = 2003*.

[HRB+06] Mark Harman, Filippo Ricca, David Binkley, Mariano Ceccato, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[HWG06] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series)*. Addison-Wesley Professional, 2006.

[Int06] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 4th edition, June 2006.

[JHo] Jhotdraw: a java gui framework for technical and structured graphics. `www.jhotdraw.org`.

[Joh77] S. C. Johnson. Lint, a c program checker. In *Computer Science Technical Report, Bell Laboratories*, NJ, USA, December 1977. Murray Hill.

[Kic96] Gregor Kiczales. Aspect-oriented programming: A position paper from the Xerox PARC aspect-oriented programming project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. Dpunkt Verlag, 1996.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[KM05] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the International Conference on Software Engineering*, 2005.

[KMT07] Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4(4640):143–162, 2007.

[KP99] Kostas Kontogiannis and Prashant Patil. Evidence driven object identification in procedural code. In *STEP '99: Proceedings of the Software Technology and Engineering Practice*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[Kri06] Jens Krinke. Mining control flow graphs for crosscutting concerns. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 334–342, Washington, DC, USA, 2006. IEEE Computer Society.

[KSG+06] Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

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

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[LFGP97] Giuseppe A. Di Lucca, Anna Rita Fasolino, Patrizia Guerra, and Silvia Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 122–129, Washington, DC, USA, 1997. IEEE Computer Society.

[LM99] A. Lai and G. Murphy. The structure of features in java code: An exploratory investigation. In *OOPSLA'99 Multi-Dimensional Separation of Concerns Workshop*, 1999.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[Mar00] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 350–359, Los Alamitos CA, 2004. IEEE Computer Society Press.

[MDM07] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[MGMD08] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE 2008), Hungary*, pages 276–291, 2008.

[MHVG08] Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In *ICFCA*, pages 289–304, 2008.

[MKK08] Kim Mens, Andy Kellens, and Jens Krinke. Pitfalls in aspect mining. In *Proceedings of WCRE '08 (15th Working Conference on Reverse Engineering)*, pages 113–122, 2008.

[MlHG06] Naouel Moha, Duc loc Huynh, and Yann-Gaël Guéhéneuc. Une taxonomie et un métamodèle pour la détection des défauts de conception. In *Langages et Modèles à Objets*, pages 201–216, 2006.

[Moh08] Naouel Moha. *DECOR : Détection et correction des défauts dans les systèmes orientés objet*. Ph.D. thesis, Université des Sciences et Technologies de Lille, August 2008.

[Mon05] Miguel Pessoa Monteiro. *Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts*. Ph.D. thesis, Universidade do Minho, July 2005.

[Moo96] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications)*, pages 235–250. ACM Press, 1996.

[Mun05] Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 15, Washington, DC, USA, 2005. IEEE Computer Society.

[NDe05] Ndepend. Website, 2005. http://www.ndepend.com/.

[NK95] Philipp Newcomb and Gordon Kotik. Reengineering procedural into object-oriented systems. In *Proceedings of WCRE (Working Conference on Reverse Engineering)*, pages 237–250. IEEE CS, 1995.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, December 1972.

[PGN08] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visuallization*, pages 77–86, New York, NY, USA, 2008. ACM.

[PMD02] Pmd. Website, June 2002. http://pmd.sourceforge.net/.

[Por80] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[RBC05] Meghan Revelle, Tiffany Broadbent, and David Coppit. Understanding concerns in software: Insights gained from two case studies. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

[RM02] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM Press.

[RM07] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3, 2007.

[RURD07] Chanchal Kumar Roy, Mohammad Gias Uddin, Banani Roy, and Thomas R. Dean. Evaluating aspect mining techniques: A case study. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 167–176, Washington, DC, USA, 2007. IEEE Computer Society.

[RW98] Spencer Rugaber and Jim White. Restoring a legacy: Lessons learned. *IEEE Software*, 15(4):28–33, July 1998.

[SB] Assia Ait Ali Slimane and Muhammad Usman Bhatti. Utilisation des services et des aspects pour la réutilisabilité du logiciel

d'un automate pour l'analyse de plasma. In *Actes de la Quatrième Journée Francophone sur le Développement du Logiciel par Aspects (JFDLPA'07)*, Toulose, France, March.

[SCRR05] Américo Sampaio, Ruzanna Chitchyan, Awais Rashid, and Paul Rayson. Ea-miner: a tool for automating aspect-oriented requirements identification. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 352–355, New York, NY, USA, 2005. ACM.

[Sem07] Semmlecode. Website, October 2007. http://semmle.com/.

[SGM00] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 154–162, Washington, DC, USA, 2000. IEEE Computer Society.

[SGP04] David Shepherd, Emily Gibson, and Lori L. Pollock. Design and evaluation of an automated aspect mining tool. In *Software Engineering Research and Practice*, pages 601–607, 2004.

[Sif98] Michael Benjamin Siff. *Techniques For Software Renovation*. Ph.D. thesis, University Of Wisconsin-Madison, May 1998.

[SLMM99] H. A. Sahraoui, H. Lounis, W. Melo, and H. Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engineering Journal*, 6(4):387–410, 1999.

[SMLD97] Houari A. Sahraoui, Walcélio Melo, Hakim Lounis, and Francois Dumont. Applying Concept Formation Methods to Object Identification in Procedural Code. In *Proceedings of ASE '97 (12th International Conference on Automated Software Engineering)*, pages 210–218. IEEE, IEEE Computer Society Press, November 1997.

[Sne96] Gregor Snelting. Reengineering of Configurations Based on Mathematical Concept Analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[Som00] Ian Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[SPPCC05] David Shepherd, Jeffrey Palm, Lori Pollock, and Mark Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 184–193, New York, NY, USA, 2005. ACM.

[SR99] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *Transactions on Software Engineering*, 25(6):749–768, November 1999.

[SS04] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[ST97] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies using Concept Analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997.

[ST98] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[SVKS02] H. Sahraoui, P. Valtchev, I. Konkobo, and S. Shen. Object identification in legacy code as a grouping problem. In *Proceedings of the 26th Computer Software and Applications Conference (COMPSAC'02)*, 2002.

[TC04a] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of WCRE 2004 (11th International Working Conference in Reverse Engineering)*, pages 112–121. IEEE Computer Society Press, November 2004.

[TC04b] Paolo Tonella and Mariano Ceccato. Migrating interface implementation to aspects. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 220–229, Washington, DC, USA, 2004. IEEE Computer Society.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167. IEEE Computer Society Press, 2000.

[TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003)*, pages 91–100. IEEE Computer Society Press, March 2003.

[TM04] Tom Tourwe and Kim Mens. Mining aspectual views using formal concept analysis. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 97–106, Washington, DC, USA, 2004. IEEE Computer Society.

[TM05] Adrian Trifu and Radu Marinescu. Diagnosing design problems in object oriented systems. In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE 2005), 7-11 November 2005, Pittsburgh, PA, USA*, pages 155–164, Los Alamitos CA, 2005. IEEE Computer Society.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

[Tri08] Mircea Trifu. Using dataflow information for concern identification in object-oriented software systems. In *Conference on Software Maintenance and Reengineering, 2008*, pages 193–202, 2008.

[TWSM94] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse enginnering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[vDK99a] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *ICSE*, pages 246–255. IEEE Press, 1999.

[vDK99b] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of ICSE '99 (21st International Conference on Software Engineering)*, pages 246–255. ACM Press, 1999.

[vEBvDT05] Remco van Engelen, Magiel Bruntink, Arie van Deursen, and Tom Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10):804–818, 2005.

[VGRH03] Petko Valtchev, David Grosser, Cyril Roume, and Mohamed Rouane Hacene. Galicia: an open platform for lattices. In *11th International Conference on Conceptual Structures (ICCS'03*, pages 241–254. Shaker Verlag, 2003.

[vM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, October 2002.

[WGH00] Eric Wong, Swapna Gokhale, and Joseph Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.

[Wig97] Theo Wiggerts. Using clustering algorithms in legacy systems remodularization. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, pages 33–43. IEEE Computer Society Press, 1997.

[Wil81] Rudolf Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, 83:445–470, September 1981.

[ZJ04] Charles Zhang and Hans-Arno Jacobsen. Prism is research in aspect mining. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 20–21, New York, NY, USA, 2004. ACM.